AFRL-IF-RS-TR-1998-99
Final Technical Report
June 1998

FORMAL VERIFICATION OF SECURITY PROPERTIES OF PRIVACY ENHANCED MAIL

Syracuse University

Shiu-Kai Chin and Dan Zhou

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-1998-99 has been reviewed and is approved for publication.

APPROVED: [illegible]
MARK GERKEN
Project Engineer

FOR THE DIRECTOR: [illegible]
NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1998	3. REPORT TYPE AND DATES COVERED Final Jun 96 - Jul 97

4. TITLE AND SUBTITLE FORMAL VERIFICATION OF SECURITY PROPERTIES OF PRIVACY ENHANCED MAIL	5. FUNDING NUMBERS C - F30602-96-1-0250 PE - 61102F PR - 2304 TA - FT WU - P1
6. AUTHOR(S) Shiu-Kai Chin and Dan Zhou	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University Department of Electrical Engineering and Computer Science Syracuse NY 13244	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTD 525 Brooks Road Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1998-99

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Mark J. Gerken, Capt, USAF/IFTD/(315) 330-2974

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The increased use of networked and distributed computing makes security a major concern. The capability to verify that a system meets its security requirements will continue to grow in importance. In particular, the capability to assign security properties to engineering structures is crucial. This work focuses on verifying the security properties of Privacy Enhanced Mail (PEM). Security properties such as non-repudiation, privacy, authentication, and integrity are defined independently of any implementation structure. PEM message structures and operations on those structures are shown to have the desired security properties. All formal definitions and proofs of security properties are done using the Higher Order Logic theorem prover. This work on PEM shows the feasibility of using formal logic and computer assisted reasoning tools to describe and verify relatively complex systems.

14. SUBJECT TERMS Computer Security, Privacy Enhanced Mail, Higher Order Logic, Formal Methods			15. NUMBER OF PAGES 128
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to describe in detail how security properties are related to secure electronic mail message formats and operations. We show how system-level security properties are satisfied by functional specifications of operations on specific message formats.

Our formal analysis is based on the Internet *Privacy Enhanced Mail* (PEM) described in four *Request for Comment* (RFC) papers: RFC 1421, RFC 1422, RFC 1423, and RFC 1424, [9, 8, 1, 7]. PEM is similar to military systems such as the National Security Agency's Multilevel Information Systems Security Initiative (MISSI). MISSI is based in part on PEM. While the message field names and structure may differ somewhat between MISSI and PEM, the analytical techniques used here are applicable to both.

We use several means of description. *Informal* descriptions are used to give an intuitive notion of behavior, properties, or requirements. These are derived from the above-cited documents. *Formal* descriptions are derived from the informal descriptions. These are intended to be *precise* descriptions of behavior which are subject to rigorous analysis. The types of analysis done includes *correctness* – e.g. ensuring requirements are met, and *behavioral properties* – e.g. security properties.

Our formal descriptions focus on:

- Structure of well-formed messages.
- Interpretation of message structures.
- Correctness of functions operating on messages.

Higher-order logic is used throughout. Verification is done using the Higher Order Logic (HOL) theorem-prover, [5].

The work described here builds on two previous efforts to formally model MISSI. The first effort by Johnson, Saydjari, and Van Tassel in [4] defines various MISSI security properties in higher-order logic. The MISSI Certificate Authority Workstation (CAW) has been modeled by Marron using a CSP (Communicating Sequential Process)-like [6] process language called *PROMELA* and the *SPIN* model checker, [10].

1.2 Network Components

The objective is to send messages securely from one local area network (LAN) to another over a wide area network (WAN) like the Internet. Components appear within the context of a WAN or LAN. Section 1.2.1 gives an overview of components which exist in the WAN. Section 1.2.2 gives an overview of components which exist within LANs.

1.2.1 WAN Components

Figure 1.1 shows two local area networks, called *enclaves* in MISSI, connected to a WAN with a *Directory System* and an *Electronic Key Management System Central Facility* (EKMS CF).

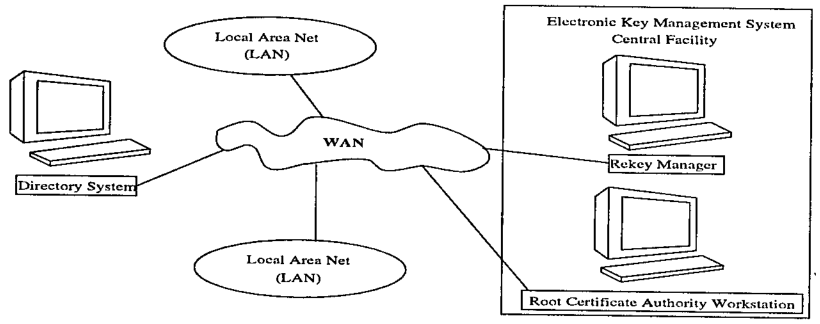

Figure 1.1 Wide Area Network Components

From Figure 1.1 we can see that the concern is with secure electronic mail *between enclaves* or LANs. Local security issues *within* a particular enclave are not addressed.

The *Directory System* functions as a "yellow-pages" for looking up people's security information such as cryptographic key information, cryptographic algorithms, the authority which has certified the authenticity of the information, and the duration or times for which the information is valid.

The *Electronic Key Management System Central Facility* serves as 1) the ultimate certification authority via the *Root Certificate Authority Workstation*, 2) support for replacing cryptographic keys (rekeying) which have expired via the *Rekey Manager*, and 3) support for *Compromised Key Lists* (CKL).

When a sender or *originator* in one enclave wishes to send email to a receiver or *recipient* in another enclave, the originator gets from the Directory System the necessary cryptographic keys and authorization to communicate with the recipient. To check if the cryptographic keys are still valid, the Compromised Key List is checked to see if the received keys are invalid because they have been compromised. As keys have finite lifetimes, user cryptographic keys must be replaced. This is done by the *Rekey Manager.*

1.2.2 LAN Components

Figure 1.2 shows the principal components within an enclave or local area network. In general, enclaves may have both trusted and untrusted workstations. The functions of the principal LAN components are illustrated by the sending of email.

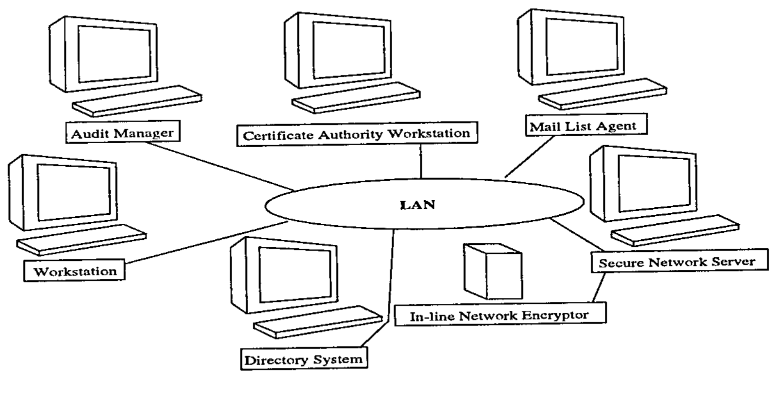

Figure 1.2 Local Area Network Components

To send email, the originator must first be *registered* or *certified* as a valid system user. This is done by the local *Certificate Authority Workstation*

(CAW). Certified users have cryptographic information and authorizations assigned to them by the CAW. Cryptographic information and authorizations are stored in data structures called *certificates.* Certificates are the means by which cryptographic information is distributed through networks.

Users have "smart cards" called *Crypto Peripherals* (CP) or *Personal Computer Memory-Card International Association* (PCMCIA) cards. Like everyday ATM cards, these cards have a PIN number known only to the user. What makes the cards smart is the information contained within them including: cryptographic algorithms, keys, and authorizations. *Type 1* cards are approved for handling classified U.S. Government information. *Type 2* cards are approved for handling sensitive but unclassified (SBU) information. The FORTEZZA card [12] is an instance of such a card. Details of its operation are not important.

A workstation with a PCMCIA card reader will take a PCMCIA card and use the cryptographic information on it for various secure email functions like encryption. Registered user• have access to a variety of MISSI functions depending on their authorizations.

The first step in sending out mail is giving the destination address. Destination addresses can be gotten from the *Directory System.* If the message is going to several recipients, i.e. is being sent to a distribution list, the message is sent to the *Mail List Agent* which forwards the message to each recipient after checking each of their credentials.

The *Secure Network Server* (SNS) serves as a guard or firewall between the enclave and the WAN. Messages from untrusted workstations within an enclave must pass through the SNS before going out on the WAN. The SNS ensures only encrypted messages go to the WAN.

Messages from trusted workstations may or may not go through the SNS. If a trusted workstation has "downgraded" the security classification of a message, this downgrade must be approved by the SNS.

Messages classified as top secret or higher must pass through the SNS and then be encrypted by an *In-line Network Encryptor* (INE) regardless of whether or not they were generated by a trusted or untrusted workstation.

1.3 Electronic Mail Scenario

A typical scenario is described by Marron in [10] as follows. Emily is in enclave A. She is registered and has a certificate with her cryptographic information authorized by the Certificate Authority Workstation in enclave

A (CAWA). Benjamin is a valid user in enclave B and has a certificate with his cryptographic information authorized by the Certificate Authority Workstation in enclave B (CAWB). Both CAWA's and CAWB's certificates are authorized by a Policy Creation Authority (PCA), and the PCA's certificate was issued by the Policy Approving Authority (PAA).

Emily wishes to send an encrypted message to Benjamin, so she does the following:

1. Computes her signature (an encrypted message based on the message text) – this is easy since she knows her own key material.
2. Electronically requests Benjamin's certificate from the Directory Service Agent (DSA). Benjamin's certificate arrives, Emily sees that it is signed by CAWB, so she requests CAWB's certificate.
3. Similarly, she next requests the PCA's certificate.
4. After receiving the PCA's certificate, she can validate it without further DSA access, since the issuers (PAA's) public key material is loaded in her FORTEZZA (Plus). She then validates the certificate for CAWB and, finally, for Benjamin.
5. Now Emily has the necessary key material to perform the public key exchange with Benjamin and mail her message.

1.4 Motivation

The security requirements placed on systems such as PEM and MISSI raise the fundamental question, *"how will we precisely understand the security requirements and by what means will we assure our designs satisfy them?"* In other words, how do we build it and how do we know it works?

The engineering view we adopt is to use techniques which answer:

1. What objects are built?
2. What are the operations on the objects?
3. How is it known if the objects are correct?

In the case of PEM and MISSI, the objects of interest are electronic mail messages. Messages have defined structures. Just as language syntax is assigned meaning by a semantic interpretation, messages have a security interpretation as well. Security functions and services are determined by the particular message type or structure.

1.5 Structure of this Report

An informal overview of security functions in general and PEM in particular is given in Chapter 2. A formal theory in higher-order logic of PEM message formats, message operations, and security properties is developed in Chapter 3. Conclusions are given in Chapter 4.

Appendix A defines the notational conventions of extended Backus-Naur Form (BNF). Appendix B is a listing of the theory defining the message structure of PEM messages in higher-order logic. Appendix C is a listing of the theory defining the operations on PEM message structures. Appendix D shows the theory applicable to MIC-CLEAR messages, i.e. messages which are transmitted without encryption or encoding but are checked for integrity. Appendix E shows the theory applicable to ENCRYPTED messages. In particular, it shows the correctness of the checks for privacy, message integrity, source authenticity, and non-deniability.

Chapter 2

Privacy Enhanced Mail

PEM adds privacy, source authentication, integrity protection, and non-repudiation services to plain text email on the Internet. PEM is documented in four *Request for Comments* (RFC) documents. RFC 1421 [9] describes message encryption, authentication procedures, and formats. RFC 1422 [8] describes certificate-based key management. RFC 1423 [1] describes algorithms. RFC 1424 [7] describes key certification.

MISSI is similar to Internet Privacy Enhanced Mail (PEM) with the exception that MISSI uses guards to protect enclaves from inappropriately releasing classified information.

2.1 Security Issues for Electronic Mail

Four key issues for secure electronic mail are identified by RFC 1421 and defined by Kaufman, Perlman, and Speciner in [2]:

- **privacy** – the ability to keep anyone but the intended recipient from reading the message.
- **authentication** – reassurance to the recipient of the identity of the sender.
- **integrity** – reassurance to the recipient that the message has not been altered since it was transmitted by the sender.
- **non-repudiation** – the ability of the recipient to prove to a third party that the sender really did send the message, i.e. the originator cannot deny sending the message.

PEM does not address all security issues. RFC 1421 identifies the following security issues *not* addressed by PEM:

- **access control** – mechanisms for restricting the use of some resource only to authorized users.
- **traffic flow confidentiality** – preventing knowledge that a message was sent.
- **address list accuracy.**
- **routing control.**
- **casual serial reuse of PCs by multiple users.**
- **assurance of message receipt and non-deniability of receipt.**
- **automatic association of acknowledgments with the messages to which they refer.**
- **message duplicate detection and replay prevention.**

In this chapter we will describe how the issues of privacy, authentication, integrity, and non-repudiation are addressed by PEM. Section 2.2 gives an overview of cryptographic functions used by PEM. Section 2.3 describes the structure of PEM messages. Section 2.4 gives examples of various PEM messages and structures. Section 2.5 describes PEM's privacy functions. Section 2.6 describes PEM's methods for source authentication. Section 2.7 describes how message integrity is checked. Section 2.8 describes mechanisms for non-repudiation.

2.2 Cryptography

Cryptography serves privacy needs by encryption. It serves source authentication and non-repudiation needs through the use of secrets. It serves integrity through message integrity codes (MIC) for secret key cryptography or digital signatures for public key cryptography.

2.2.1 Types of Cryptographic Functions

There are three kinds of cryptographic functions: secret key functions, public key functions and hash functions. Public key cryptography uses two keys. Secret key cryptography use one key. Hash functions uses no keys.

Secret Key Cryptography

In *Secret key* or *symmetric* cryptography, the same key s is used for both encryption and decryption, as shown in Figure 2.1. Ciphertext is obtained by applying the encryption function to both plaintext and the secret key. To retrieve the original plaintext, decryption function is applied to the ciphertext and the same secret key. A message m encrypted with secret key s is denoted as $[m]_s$.

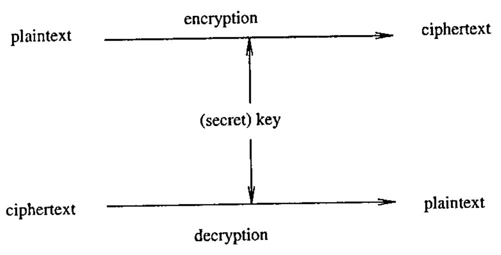

Figure 2.1 Secret Key Cryptography

Ideally secret key cryptography has following property: a message encrypted with secret key k can only be retrieved (decrypted) with the same secret key. When an initial vector(IV) is utilized in the cryptographic algorithm, it must be the same for both encryption and decryption. This can be formalized as:

$$\begin{aligned}&\forall msg\ key\ IV.\\ &\quad (decryptS\ (encryptS\ msg\ key\ IV)\ key\ IV\ = msg)\ \wedge\\ &\quad (\forall msg2\ key2.\ (decryptS\ msg2\ key\ IV =\\ &\qquad decryptS\ msg2\ key2\ IV) = key = key2)\end{aligned} \tag{2.1}$$

The secret key scheme can be used to generate a fixed-length cryptographic checksum associated with a message, as shown in Figure 2.2; this message integrity code (MIC) can be used to check the integrity of the message sent along with it (see section 2.2.4).

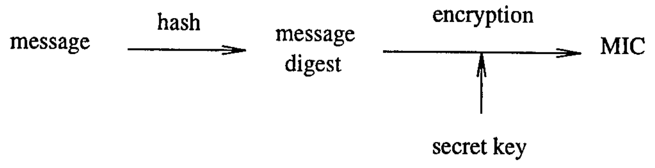

Figure 2.2 Message Integrity Code

Public Key Cryptography

In *public key* or *asymmetric* cryptography, each individual has a pair of keys: a private key d only known to the owner, and a corresponding public key e that is accessible by the world. The public key is used for encryption and the private key is used for decryption. This is shown in Figure 2.3. A message m encrypted using public key e is denoted as $\{m\}_e$.

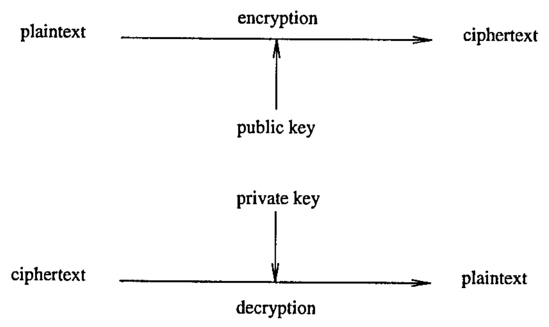

Figure 2.3 Public Key Cryptography

Public key cryptography has following property: a message encrypted with public key e_k can only be retrieved (decrypted) with an unique private key d_k; on the other hand, a message encrypted with private key d_k can only be retrieved (decrypted) with the unique public key e_k. This can be formalized as:

$$\forall msg\ eKEY\ dKEY.$$
$$((decryptP\ (encryptP\ msg\ eKEY)\ dKEY\ = msg)\ = \qquad (2.2)$$

$$\begin{aligned}&(encryptP\ (decryptP\ msg\ dKEY)\ eKEY\ = msg)) \wedge\\&((decryptP\ (encryptP\ msg\ eKEY)\ dKEY\ = msg) \supset\\&((\forall dk.\ (decryptP\ (encryptP\ msg\ eKEY)\ dk\ = msg) \supset dk = dKEY) \wedge\\&(\forall ek.\ (encryptP\ (decryptP\ msg\ dKEY)\ ek\ = msg) \supset ek = eKEY))\end{aligned} \tag{2.3}$$

Public key cryptography can be used to generate signature on any message. The signature can be verified by anyone who knows the public key of the signer, and can only be generated by the one who knows the corresponding private key. This is shown in Figure 2.4. These two properties can be formalized as follows:

$$\begin{aligned}&\forall m1\ m2\ dkey1\ dkey2.\ (sign\ m1\ dkey1 = sign\ m2\ dkey2)\\&\quad \supset (m1 = m2) \wedge (dkey1 = dkey2)\end{aligned} \tag{2.4}$$

$$\begin{aligned}&\forall msg\ eKEY\ dKEY.\ verify\ msg\ (sign\ msg\ dKEY)\ eKEY \supset\\&\quad (\forall m1\ m2.verify\ m1\ m2\ eKEY = (m2 = sign\ m1\ dKEY))\end{aligned} \tag{2.5}$$

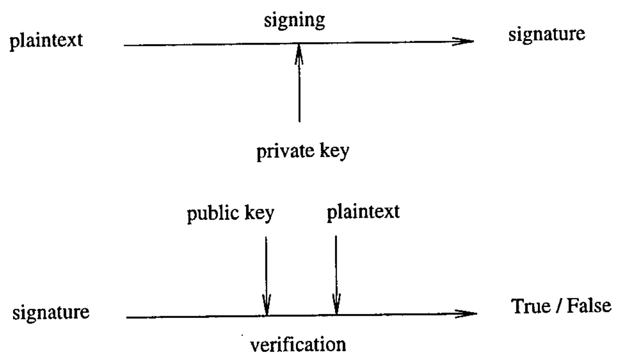

Figure 2.4 Signature

The counterpart of MICs for public key cryptography are digital signatures as shown in Figure 2.5. They are used to check integrity.

Hash Functions

Hash functions are message digests or one-way transformations. A cryptographic hash function is a mathematical transformation that takes a mes-

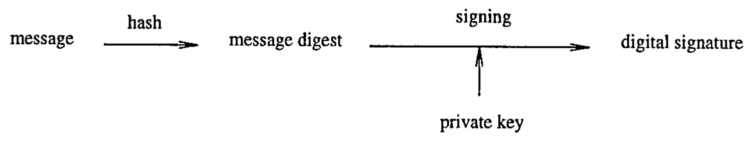

Figure 2.5 Digital Signature

sage of arbitrary length and computes from it a fixed length number.

Hash functions have the following properties:

- If $h(m_0)$ denotes the hash of the message m_0, there is no substantially easier way to find an m whose hash is $h(m_0)$ without going through all values of m to search for $h(m_0)$.
- It is computationally infeasible to find two values of m which hash to the same value.

Essentially, hash functions behave like one-to-one functions, i.e.,

$$\forall m\, m'.\, h(m) = h(m') \supset m = m' \tag{2.6}$$

2.2.2 Privacy

Privacy is obtained through encryption. If Emily wants to send Benjamin a mail that only Benjamin can read, she will choose a random secret key S to be used only for encrypting that one message m. She encrypts the message with S to get $[m]_S$, encrypts S with Benjamin's public key e_B to get $\{S\}_{e_B}$(if public key cryptography is used) or with the secret key she shares with Benjamin K_{EB} to get $[S]_{K_{EB}}$(if secret key cryptography is used), and transmits both to Benjamin.

Privacy in PEM is gotten by using any of the following cryptographic functions: DES-CBC for secret key encryption of messages; DES-EDE for secret key encryption of Data Encryption Keys (DEKs); DES-ECB for secret key encryption of DEKs; RSA for public key encryption of DEKs and signatures. Summaries of each of the encryption algorithms mentioned here are found in [2].

2.2.3 Authentication

Authentication verifies the identity of the communicating party. Encryption is used to prove the knowledge of secrets, hence to verify identities. The means for doing so are variations on a *challenge/response* protocol. A challenge is issued by the party wishing to verify the identity of the other principal. The principal, whose identity is being checked, issues a response based on the use of a secret key or public key cryptography.

In secret key cryptography, if Emily wants to verify the identity of Benjamin, she issues a challenge, a random picked number r, and sends it to Benjamin. Benjamin encrypts the r with the the secret key K_{EB} he shares with Emily and sends it back to Emily. Emily decrypts the response with K_{EB} and checks to see if she got back r (see Figure 2.6).

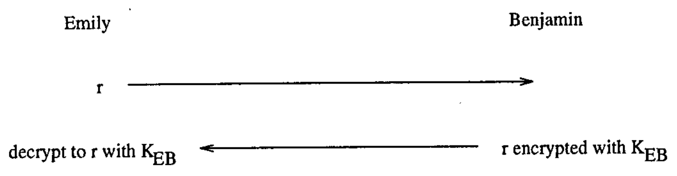

Figure 2.6 Secret Key Authentication

If public key cryptography is used, Emily chooses a random number r, encrypts it with Benjamin's public key e_B and sends the result to Benjamin. Benjamin proves he knows his private key d_B by decrypting the message and sending r back to Emily (see Figure 2.7).

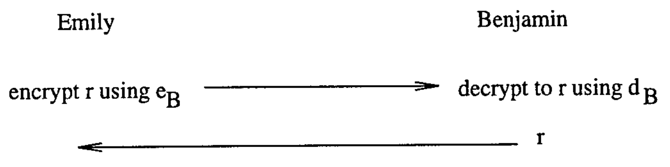

Figure 2.7 Public Key Authentication

2.2.4 Integrity

Integrity of a message is maintained by using either a MIC (in secret key cryptography) or a digital signature (in public key cryptography) shown in Figure 2.2 and Figure 2.5.

For secret key cryptography, a MIC is computed by using a secret key with a known checksum algorithm. It is included as part of the header sent along with the message to the recipients. The recipients compute the MIC for the message they receive and compare it to the MIC received in the header. If the MICs match, then the message is genuine (see Figure 2.8).

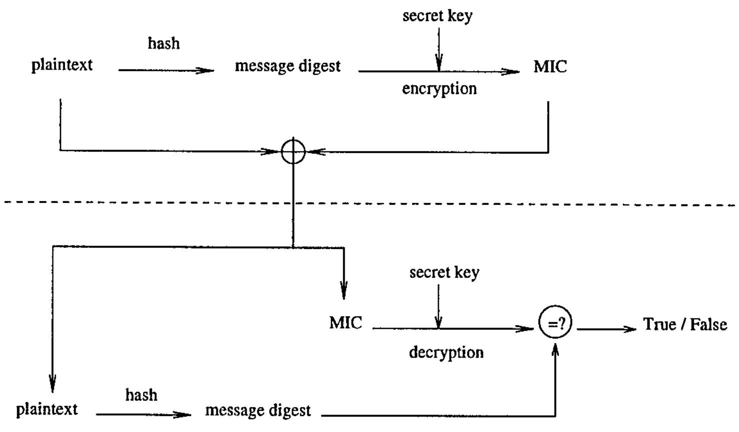

Figure 2.8 Integrity Check using a MIC

For public key cryptography, integrity is protected by digital signatures. If Emily wants to send Benjamin a message which is integrity protected, she generates the digital signature of the message using her private key, and send it along with the message to Benjamin. When Benjamin receives the message with its digital signature, he verifies the digital signature with Emily's public key (see Figure 2.9).

Hash functions are used with public keys for integrity protection (see Figure 2.9). Signing a message digest is much quicker than signing a message itself. When the signature of the message digest is sent with the message to recipients, the recipients generate the message digest from the message, and verify the signature of the digest to check the integrity of the message.

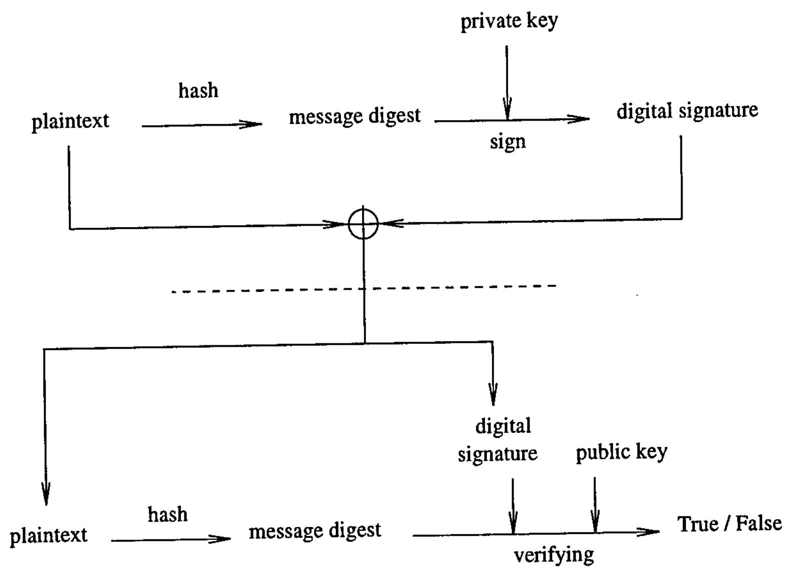

Figure 2.9 Integrity Check using a Digital Signature

2.2.5 Non-repudiation

Non-repudiation is the ability of the recipient to prove to a third party that the sender really did send the message. It comes automatically with public key cryptography as only the person who knows the private key can generate the signature. Comparing the message digest with the signature decrypted using originator's public key is all that is required.

2.3 Structure of PEM Messages

This section describes the structure of a PEM message. It is excerpted from RFC 1421, *Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*, [9]. Included is an additional message type, *CRL-retrieval request* as described in RFC 1424, *Key Certification and Related Services*, [7].

The notation used is augmented Backus-Naur Form (BNF) as described in RFC 822, [3]. A full description of the augmented BNF is in Appendix

A.

Figure 2.10 defines the top-level structure of a PEM message. The top-level structure includes:

- A Pre-Encapsulation Boundary (preeb):
  `-----BEGIN PRIVACY-ENHANCED MESSAGE-----`
- A PEM header (pemhdr) containing encryption information.
- A carriage-return-linefeed (CRLF) with the message text (pemtext), if any.
- A Post-Encapsulation Boundary (posteb):
  `-----END PRIVACY-ENHANCED MESSAGE-----`

```
; PEM BNF representation, using RFC 822 notation.

; imports field meta-syntax (field, field-name, field-body,
; field-body-contents) from RFC-822, sec. 3.2
; imports DIGIT, ALPHA, CRLF, text from RFC-822
; Note: algorithm and mode specifiers are officially defined
; in RFC 1423

<pemmsg> ::= <preeb>
             <pemhdr>
             [CRLF <pemtext>]    ; absent for CRL message
             <posteb>

<preeb> ::= "-----BEGIN PRIVACY-ENHANCED MESSAGE-----" CRLF
<posteb> ::= "-----END PRIVACY-ENHANCED MESSAGE-----" CRLF / <preeb>

<pemtext> ::= <encbinbody>       ; for ENCRYPTED or MIC-ONLY messages
            / *(<text> CRLF)     ; for MIC-CLEAR

<pemhdr> ::= <normalhdr> / <crlhdr>
```

Figure 2.10 Top-Level PEM Message Structure

A template of an encapsulated message taken from RFC 1421, [9] is shown below in Figure 2.11. The message components **<pemhdr>** and **<pemtext>** are the encapsulated header and encapsulated text portions of the message. These are described below.

Two types of encryption keys are used in PEM as reported in RFC 1421, [9].

```
Pre-Encapsulation Boundary (Pre-EB)
     -----BEGIN PRIVACY-ENHANCED MESSAGE-----

Encapsulated Header Portion
     (Contains encryption control fields inserted in plaintext.
     Examples include "DEK-Info:" and "Key-Info:".
     Note that, although these control fields have line-oriented
     representations similar to RFC 822 header fields, the set
     of fields valid in this context is disjoint from those used
     in RFC 822 processing.)

Blank Line
     (Separates Encapsulated Header from subsequent
     Encapsulated Text Portion)

Encapsulated Text Portion
     (Contains message data encoded as specified.)

Post-Encapsulation Boundary (Post-EB)
     -----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.11 Encapsulated Message Format

- Data Encryption Keys (DEKs) are used for encrypting message text and for message integrity codes (MICs). These keys are generated on a per-message basis with no prior pre-distribution.
- Interchange Keys (IKs) are used to encrypt DEKs for transmission within messages. IKs are used over a period of time. They are typically the secret or public keys of principals depending on whether secret or public key encryption is used.

2.3.1 Encapsulated Header Portion

The header portion of the message has the encryption control information necessary to decrypt the encapsulated message text portion of a PEM message. Its format is defined by RFC 1421. Its BNF description is in Figure 2.12.

There are two types of headers:

- *normal headers* `<normalhdr>` – used for messages that are not requests related to certificate revocation lists (CRLs).

- *headers for CRLs* <crlhdr> – used for messages related to CRLs.

```
<normalhdr> ::= <proctype>
            <contentdomain>
            [<dekinfo>]          ; needed if ENCRYPTED
            (1*(<origflds> *<recipflds>)) ; symmetric case --
                          ; recipflds included for all proc types
            / ((1*<origflds>) *(<recipflds>)) ; asymmetric case --
                          ; recipflds included for ENCRYPTED proc type

<crlhdr> ::= <proctype>
            1*(<crl> [<cert>] *(<issuercert>))

<asymmorig> ::= <origid-asymm> / <cert>

<origflds> ::= <asymmorig> [<keyinfo>] *(<issuercert>)
               <micinfo>                        ; asymmetric
               / <origid-symm> [<keyinfo>]      ; symmetric

<recipflds> ::= <recipid> <keyinfo>
```

Figure 2.12 PEM Header Structure

Normal Headers

Normal headers contain:

- *process type* <proctype> – the version number of PEM being used and the type of PEM message. In this case, version 4 is the only possibility. PEM message types can be ENCRYPTED, MIC-ONLY, MIC-CLEAR, CRL, or CRL-RETRIEVAL-REQUEST. See Figure 2.13.

- *content domain* <contentdomain> – the type of mail message, in this case the only possibility is RFC822 which identifies it as an ARPA Internet text message. See Figures 2.13 and 2.15.

- *data encrypting key information* <dekinfo> – required for ENCRYPTED messages. See Figures 2.13 and 2.15.

- One or more *originator fields* <origflds> with zero or more *recipient fields* <recipflds>. The required fields depend on whether secret or public key cryptography is used. See Figures 2.13 and 2.14.

```
; definitions for PEM header fields

<proctype> ::= "Proc-Type" ":" "4" "," <pemtypes> CRLF
<contentdomain> ::= "Content-Domain" ":" <contentdescrip> CRLF
<dekinfo> ::= "DEK-Info" ":" <dekalgid> [ "," <dekparameters> ] CRLF
<symmid> ::= <IKsubfld> "," [<IKsubfld>] "," [<IKsubfld>]
<asymmid> ::= <IKsubfld> "," <IKsubfld>
<origid-asymm> ::= "Originator-ID-Asymmetric" ":" <asymmid> CRLF
<origid-symm> ::= "Originator-ID-Symmetric" ":" <symmid> CRLF
<recipid> ::= <recipid-asymm> / <recipid-symm>
<recipid-asymm> ::= "Recipient-ID-Asymmetric" ":" <asymmid> CRLF
<recipid-symm> ::= "Recipient-ID-Symmetric" ":" <symmid> CRLF
<cert> ::= "Originator-Certificate" ":" <encbin> CRLF
<issuercert> ::= "Issuer-Certificate" ":" <encbin> CRLF
<micinfo> ::= "MIC-Info" ":" <micalgid> "," <ikalgid> ","
              <asymsignmic> CRLF
<keyinfo> ::= "Key-Info" ":" <ikalgid> "," <micalgid> ","
              <symencdek> "," <symencmic> CRLF     ; symmetric case
              / "Key-Info" ":" <ikalgid> "," <asymencdek>
              CRLF                                 ; asymmetric case
<crl> ::= "CRL" ":" <encbin> CRLF
<pemtypes> ::= "ENCRYPTED" / "MIC-ONLY" / "MIC-CLEAR" / "CRL"
               / "CRL-RETRIEVAL-REQUEST"
```

Figure 2.13 PEM Header Fields

- Secret (symmetric) key case: the `<origflds>` consists of the originator's id `<origid-symm>` and optional key information `<keyinfo>`. Id's typically look like: `chin@cat.syr.edu` with additional information on interchange keys (IKs). See Figures 2.12 and 2.13.
- Public (asymmetric) key case: the `<origflds>` consists of: 1) the asymmetric originator's id `<asymmorig>` which is either the asymmetric originator's id `<origid-asymm>` (as in the secret key case) or the certificate `<cert>` of the originator; 2) optional key information `<keyinfo>`; 3) zero or more issuer certificates `<issuercert>`; and 4) message integrity code `<micinfo>` information. See Figures 2.13, 2.14, and 2.15. Details on certificates are in Section 2.6.

CRL Headers

CRL headers contain:

- *process type* – same as for normal headers.
- at least one CRL with an optional certificate and zero or more issuer certificates. See Figure 2.13.

Certificates are used to authenticate principals. Details are in Section 2.6.

2.3.2 Encapsulated Text Portion

An important distinction is to be made between *encoded* versus *encrypted* messages. *Encoded* messages are those which have been modified in such a way so that there are no "funny characters" and no lines which are too long which would cause any mail system to modify the message contents. An example of this is the UNIX *uuencode* program. Of course, such encodings must be readily reversible so that the un-encoded text can be retrieved, e.g. the UNIX *uudecode* program. *Encrypted* messages are messages which have been processed using a cryptographic algorithm which of course, should only be reversible by those having the proper keys.

Table 2.1 gives the encoding used by PEM. The encoding works as follows:

- PEM sends encoded information 32-bits at a time which corresponds to four *8-bit* encoded characters.
- The four encoded *8-bit* characters are derived from four *6-bit* inputs. The six input bits have a range of possible values from 0_{10} to 63_{10} – 000000_2 to 111111_2.
- Each 6-bits is encoded as an ASCII character as shown in Table 2.1. For example, 000000_2 is encoded as ASCII character `A`.
- Each ASCII character is sent out as an *8-bit* quantity – *7-bits* representing the character plus one bit for parity (the most-significant bit). For example, `A` has an *8-bit* hex encoding 41_{16} or 01000001_2. This can be sent as $P1000001_2$ where P is the parity bit. The subset of ASCII characters used falls in the range at or below $7A_{16}$, so the entire subset can be represented with *7-bits* plus one bit for parity.
- Finally, *four* encoded *6-bit* characters (*24-bits*) are sent at a time as a 32-bit word. If the data are not a multiple of *6-bits*, the data are extended to the next multiple of *6-bits* by adding 0s as *padding bits*.

Table 2.1 PEM 6-Bit Encoding

value_{10}	character	ASCII representation
0	A	41 hex
⋮	⋮	⋮
25	Z	5A hex
26	a	61 hex
⋮	⋮	⋮
51	z	7A hex
52	0	30 hex
⋮	⋮	⋮
61	9	39 hex
62	+	2B hex
63	/	2F hex
padding	=	3D hex

If the data are not a multiple of four characters (*24-bits*), *padding* characters are sent. Padding *characters* are encoded as ASCII =, i.e. $3D_{16}$ or $P0111101_2$.

Figure 2.14 shows BNF form of the encoded binary characters, `<encbinchar>`. `<encbinchar>` are the upper and lower case letters – `ALPHA`; the digits 0 through 9 – `DIGIT`; and the characters `+`, `/`, and `=`.

A group of encoded binary characters `<encbingrp>` is exactly four encoded binary characters `4*4<encbinchar>`. A body of encoded binary character groups is zero or more *lines* of up to 16 character groups or *64 characters per line* – `*(16*16<encbingrp> CRLF) [1*16<encbingrp> CRLF]`. This can be seen in the example messages which follow.

2.4 Examples of PEM Message Types

There are five types of PEM messages – 1) *ENCRYPTED*, 2) *MIC-CLEAR*, 3) *MIC-ONLY*, 4) *CRL*, and 5) *CRL-RETRIEVAL-REQUEST*. ENCRYPTED, MIC-CLEAR, and MIC-ONLY messages have secret key and public key variants.

```
<encbinchar> ::= ALPHA / DIGIT / "+" / "/" / "="
<encbingrp> ::= 4*4<encbinchar>
<encbin> ::= 1*<encbingrp>
<encbinbody> ::= *(16*16<encbingrp> CRLF) [1*16<encbingrp> CRLF]
<IKsubfld> ::= 1*<ia-char>
; Note: "," removed from <ia-char> set so that Orig-ID and Recip-ID
; fields can be delimited with commas (not colons) like all other
; fields
<ia-char> ::=  DIGIT / ALPHA / "'" / "+" / "(" / ")" /
               "." / "/" / "=" / "?" / "-" / "@" /
               "%" / "!" / '"' / "_" / "<" / ">"
<hexchar> ::= DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
                                                ; no lower case
```

Figure 2.14 Character Descriptions

ENCRYPTED messages indicate their message bodies are encrypted. MIC-ONLY messages are those whose messages are encoded but *not* encrypted and have a MIC computed as an integrity check. MIC-CLEAR messages are those whose messages are neither encoded nor encrypted and have a MIC computed as an integrity check. CRL-RETRIEVAL-REQUEST messages have no message but are used to request CRLs. CRL messages store CRLs or reply to CRL retrieval requests.

2.4.1 ENCRYPTED

Public Key Variant

Table 2.2 shows the format of PEM messages which are encrypted using asymmetric (public) keys, [2]. Figure 2.16 is an example message taken from RFC 1421. Figure 2.17 shows the processing of PEM message on sender side, Figures 2.18, 2.19 and 2.20 show the processing of PEM message on receiver side.

Secret Key Variant

Table 2.3 shows the format of PEM messages which are encrypted using symmetric (secret) keys, [2]. Figure 2.21 is an example message taken from RFC 1421.

Table 2.2 Encrypted, Public Key PEM Message Format

`-----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4, ENCRYPTED`	type of PEM message (version,type)
`Content-Domain: RFC822`	message form
`DEK-Info: DES-CBC,` *16 hex digits*	message encryption algorithm, IV
`Originator-Certificate:` *cybercrud*	sender's encoded certificate (optional)
`Originator-ID-Asymmetric:` *cybercrud,number*	sender ID (present only if sender's certificate not present)
`Key-Info: RSA,`*cybercrud*	key-info for CC'd sender (if needed)
`Issuer-Certificate:` *cybercrud* ⋮	sequence of zero or more CA certificates (possibly whole chain from the sender's certificate to the IPRA's)
`MIC-Info: RSA-MD`*x*`,RSA,`*cybercrud*	message digest algorithm, message digest
	encryption algorithm, encoded encrypted MIC
`Recipient-ID-Asymmetric:` *cybercrud,number* `Key-Info: RSA,` *cybercrud* ⋮	For each recipient: recipient ID (encoded X.500 name of CA that signed certificate, certificate serial number); key-info for recipient
	Blank line
cybercrud	encoded encrypted message
`-----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

Table 2.3 Encrypted, Secret Key PEM Message Format

`-----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4, ENCRYPTED`	type of PEM message (version,type)
`Content-Domain: RFC822`	message form
`DEK-Info: DES-CBC,` *16 hex digits*	message encryption algorithm, IV
`Originator-ID-Symmetric:` *entity identifier, issuing authority, version/expiration*	sender ID
`Recipient-ID-Symmetric:` *entity identifier, issuing authority, version/expiration* `Key-Info: RSA,` *cybercrud* ⋮	For each recipient: recipient ID; key-info for recipient
	Blank line
cybercrud	encoded encrypted message
`-----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

```
; This specification defines one value ("RFC822") for
; <contentdescrip>: other values may be defined in future in
; separate or successor documents
;
<contentdescrip> ::= "RFC822"

; Addendum to PEM BNF representation, using RFC 822 notation
; Provides specification for official PEM cryptographic algorithms,
; modes, identifiers and formats.

; Imports <hexchar> and <encbin> from RFC [1421]

   <dekalgid> ::= "DES-CBC"
   <ikalgid>  ::= "DES-EDE" / "DES-ECB" / "RSA"
   <sigalgid> ::= "RSA"
   <micalgid> ::= "RSA-MD2" / "RSA-MD5"

   <dekparameters> ::= <DESCBCparameters>
   <DESCBCparameters> ::= <IV>
   <IV> ::= <hexchar16>

   <symencdek> ::= <DESECBencDESCBC> / <DESEDEencDESCBC>
   <DESECBencDESCBC> ::= <hexchar16>
   <DESEDEencDESCBC> ::= <hexchar16>

   <symencmic> ::= <DESECBencRSAMD2> / <DESECBencRSAMD5>

   <DESECBencRSAMD2> ::= 2*2<hexchar16>
   <DESECBencRSAMD5> ::= 2*2<hexchar16>

   <asymsignmic> ::= <RSAsignmic>
   <RSAsignmic> ::= <encbin>

   <asymencdek> ::= <RSAencdek>
   <RSAencdek> ::= <encbin>

   <hexchar16> ::= 16*16<hexchar>
```

Figure 2.15 PEM Cryptographic Algorithms, Modes, and Identifiers

2.4.2 MIC-ONLY or MIC-CLEAR

Public Key Variant

Table 2.4 shows the format of MIC-ONLY and MIC-CLEAR messages using public keys. Figure 2.22 is an example of a MIC-ONLY message. MIC-ONLY messages encode their messages as described in Section 2.1. MIC-

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator-Certificate:
 MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDzAN
 BgNVBAsTBk5PVEFSWTAeFw05MTA5MDQxODM4MTdaFw05MzA5MDMxODM4MTZaMEUx
 CzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5jLjEU
 MBIGA1UEAxMLVGVzdCBVc2VyIDEwWTAKBgRVCAEBAgICAANLADBIAkEAwHZHl7i+
 yJcqDtjJCowzTdBJrdAiLAnSC+CnnjOJELyuQiBgkGrgIh3j8/x0fM+YrsyF1u3F
 LZPVtzlndhYFJQIDAQABMA0GCSqGSIb3DQEBAgUAA1kACKr0PqphJYw1j+YPtcIq
 iW1FPuN5jJ79Khfg7ASFxskYkEMjRNZV/HZDZQEhtVaU7Jxfzs2wfX5byMp2X3U/
 5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Key-Info: RSA,
 I3rRIGXUGWAF8js5wCzRTkdh034PTHdRZY9Tuvm03M+NM7fx6qc5udixps2Lng0+
 wGrtiUm/ovtKdinz6ZQ/aQ==
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDTAL
 BgNVBAsTBFRMQ0EwHhcNOTEwOTAxMDgwMDAwWhcNOTIwOTAxMDc1OTU5WjBRMQsw
 CQYDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4xDzAN
 BgNVBAsTBkJldGEgMTEPMA0GA1UECxMGTk9UQVJZMHAwCgYEVQgBAQICArwDYgAw
 XwJYCsnp6lQCxYykNlODwutF/jMJ3kL+3PjYyHOwk+/9rLg6X65B/LD4bJHt05XW
 cqAz/7R7XhjYCm0PcqbdzoACZtIlETrKrcJiDYoP+DkZ8k1gCk7hQHpbIwIDAQAB
 MA0GCSqGSIb3DQEBAgUAA38AAICPv4f9Gx/tY4+p+4DB7MV+tKZnvBoy8zgoMGOx
 dD2jMZ/3HsyWKWgSF0eH/AJB3qr9zosG47pyMnTf3aSy2nB07CMxpUWRBcXUpE+x
 EREZd9++32ofGBIXaialnOgVUn00zSYgugiQ077nJLDUj0hQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
 UdFJR8u/TIGhfH65ieewe210W4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
 AjRFbHoNPzBuxwm0AFeA0HJszL4yBvhG
Recipient-ID-Asymmetric:
 MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
 LjEPMA0GA1UECxMGQmV0YSAxMQ8wDQYDVQQLEwZOT1RBUlk=,
 66
Key-Info: RSA,
 O6BS1ww9CTyHPtS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
 7x0Z3Jx2vTAh0YHMcqqCjA==

qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.16 Example ENCRYPTED Message (Public Key Case)

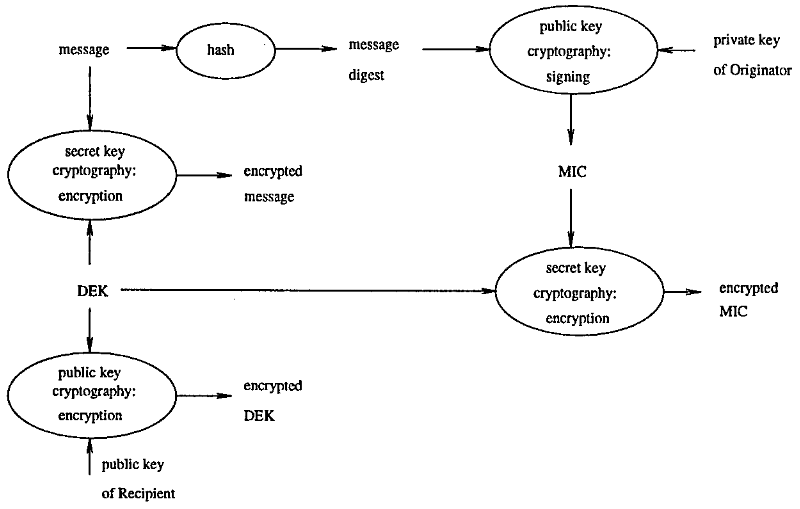
message
hash
message digest
public key cryptography: signing
private key of Originator
secret key cryptography: encryption
encrypted message
MIC
DEK
secret key cryptography: encryption
encrypted MIC
public key cryptography: encryption
encrypted DEK
public key of Recipient

Figure 2.17 Processing of PEM message on sender side: *ENCRYPTED*

Table 2.4 MIC-ONLY or MIC-CLEAR Public Key Format

`----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4, MIC-ONLY or MIC-CLEAR`	type of PEM message (version,type)
`Content-Domain: RFC822`	message form
`Originator-Certificate:` *cybercrud*	sender's encoded certificate (optional)
`Originator-ID-Asymmetric:` *cybercrud,number*	sender ID (present only if sender's certificate not present)
`Issuer-Certificate:` *cybercrud* ⋮	sequence of zero or more CA certificates (possibly whole chain from the sender's certificate to the IPRA's)
`MIC-Info: RSA-MD`x`,RSA,` *cybercrud*	message digest algorithm, message digest encryption algorithm, encoded encrypted MIC
	Blank line
message	message (encoded if MIC-ONLY)
`----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

CLEAR messages do not use encoding.

Secret Key Variant

Table 2.5 shows the format of MIC-ONLY and MIC-CLEAR messages using secret keys. MIC-ONLY messages have their message contents encoded as described in Section 2.1. MIC-CLEAR messages do not use encoding.

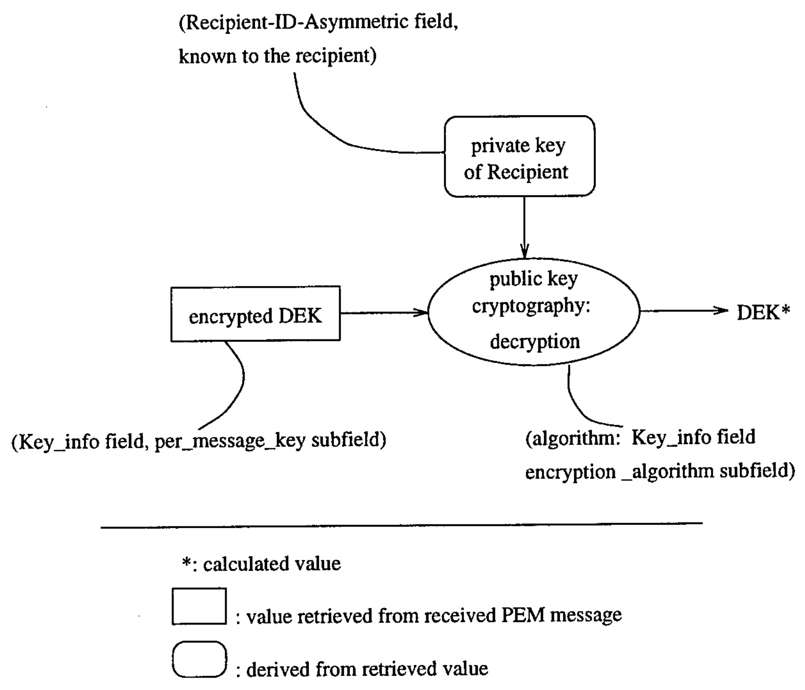
(Recipient-ID-Asymmetric field, known to the recipient)
private key of Recipient
encrypted DEK
public key cryptography: decryption
DEK*
(Key_info field, per_message_key subfield)
(algorithm: Key_info field encryption _algorithm subfield)
*: calculated value
: value retrieved from received PEM message
: derived from retrieved value

Figure 2.18 Processing of PEM message on receiver side - Retrieve DEK: *ENCRYPTED*

2.4.3 CRL-RETRIEVAL-REQUEST

Table 2.6 gives the format of a CRL-RETRIEVAL-REQUEST message. Figure 2.23 is an example from RFC 1421 of such a request.

2.4.4 CRL

Table 2.7 gives the format for CRL messages. Figures 2.24 and 2.25 illustrate CRL storage request and retrieval reply messages.

Table 2.5 MIC-ONLY and MIC-CLEAR Secret Key Format

`-----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4, ENCRYPTED`	type of PEM message (version,type)
`Content-Domain: RFC822`	message form
`Originator-ID-Symmetric:` *entity identifier, issuing authority, version/expiration*	sender ID
`Recipient-ID-Symmetric:` *entity identifier issuing authority, version/expiration* `Key-Info: RSA,` *cybercrud* ⋮	For each recipient: recipient ID; key-info for recipient
	Blank line
message	message (encoded if MIC-ONLY)
`-----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

Table 2.6 CRL-RETRIEVAL-REQUEST Format

`-----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4,CRL-RETRIEVAL-REQUEST`	type of PEM message (version,type)
`Issuer:` *cybercrud* ⋮	for each CRL requested: the encoded X.500 name of the issuing CA
`-----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

Table 2.7 CRL Format

`-----BEGIN PRIVACY-ENHANCED MESSAGE-----`	pre-encapsulation boundary
`Proc-Type: 4,CRL`	type of PEM message (version,type)
`CRL:` *cybercrud* `Originator-Certificate:` *cybercrud* ⋮	For each CRL retrieved: encoded X.509 format CRL; encoded X.509 certificate of the CA that issued the CRL
`-----END PRIVACY-ENHANCED MESSAGE-----`	post-encapsulation boundary

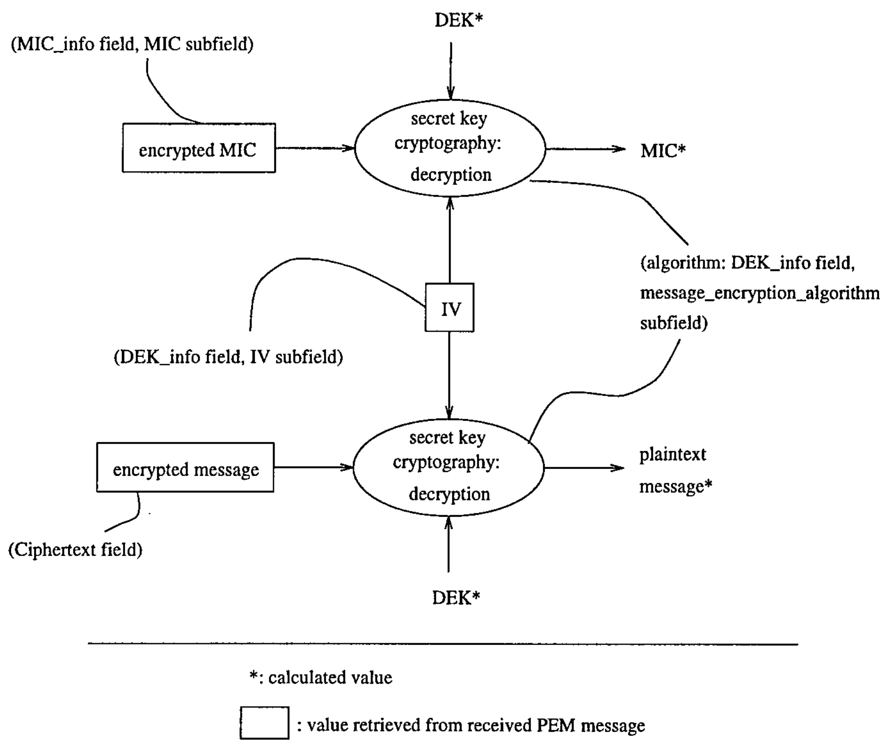
DEK*
(MIC_info field, MIC subfield)
encrypted MIC
secret key cryptography: decryption
MIC*
(algorithm: DEK_info field, message_encryption_algorithm subfield)
IV
(DEK_info field, IV subfield)
encrypted message
secret key cryptography: decryption
plaintext message*
(Ciphertext field)
DEK*
*: calculated value
: value retrieved from received PEM message

Figure 2.19 Processing of PEM message on receiver side - Retrieve plaintext message and MIC: *ENCRYPTED*

2.5 Privacy in PEM

The cryptographic algorithms, modes, and identifiers for PEM are defined in RFC 1423, [1] along with the content description in RFC 1421, [9]. The structural definition in BNF form appears in Figure 2.15.

The cryptographic functions used in PEM are:

- DES-CBC – (*Data Encryption Standard Cipher Block Chaining*) for secret key encryption of messages.
- DES-EDE – (*DES encrypt-decrypt-encrypt*) for secret key encryption of DEKs.

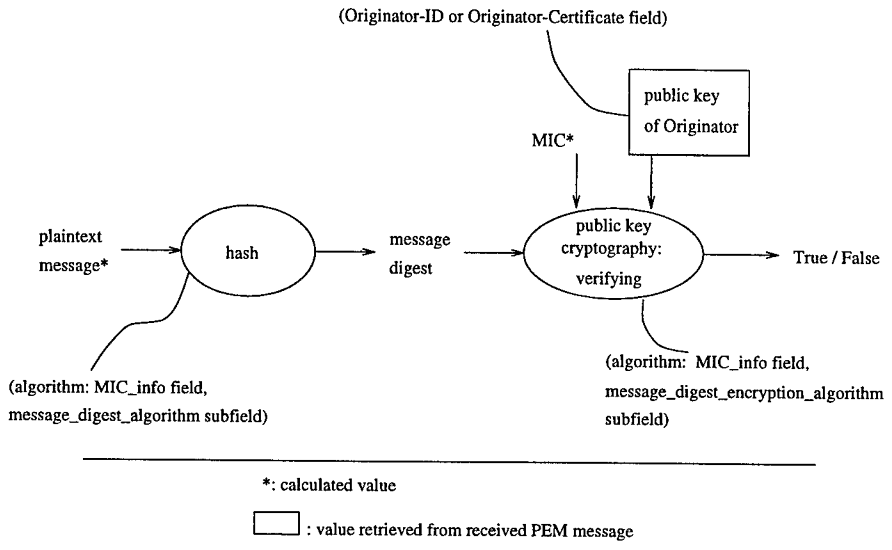
(Originator-ID or Originator-Certificate field)
public key of Originator
MIC*
plaintext message*
hash
message digest
public key cryptography: verifying
True / False
(algorithm: MIC_info field, message_digest_algorithm subfield)
(algorithm: MIC_info field, message_digest_encryption_algorithm subfield)
*: calculated value
: value retrieved from received PEM message

Figure 2.20 Processing of PEM message on receiver side - Verify digital signature: *ENCRYPTED*

- DES-ECB – (*DES electronic code book*) for secret key encryption of DEKs.
- RSA – (*Rivest, Shamir, and Adleman*) for public key encryption of DEKs and signatures.
- RSA-MD2 – (*RSA message digest 2*) for secret key computation of message integrity codes.
- RSA-MD5 – (*RSA message digest 5*) for secret key computation of message integrity codes.

2.6 Authentication in PEM

2.6.1 Certificates

Authentication in PEM is done using *certificates.* Certificates are data structures which contain the public information of users. This public information includes:

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,F8143EDE5960C597
Originator-ID-Symmetric: linn@zendia.enet.dec.com,,
Recipient-ID-Symmetric: linn@zendia.enet.dec.com,ptf-kmc,3
Key-Info: DES-ECB,RSA-MD2,9FD3AAD2F2691B9A,
          B70665BB9BF7CBCDA60195DB94F727D3
Recipient-ID-Symmetric: pem-dev@tis.com,ptf-kmc,4
Key-Info: DES-ECB,RSA-MD2,161A3F75DC82EF26,
          E2EF532C65CBCFF79F83A2658132DB47

LLrHB0eJzyhP+/fSStdW8okeEnv47jxe7SJ/iN72ohNcUk2jHEUSoH1nvNSIWL9M
8tEjmF/zxB+bATMtPjCUWbz8Lr9wloXIkjHUlBLpvXR0UrUzYbkNpk0agV2IzUpk
J6UiRRGcDSvzrsoK+oNvqu6z7Xs5Xfz5rDqUcMlK1Z6720dcBWGGsDLpTpSCnpot
dXd/H5LMDWnonNvPCwQUHt==
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.21 Example ENCRYPTED Message (Secret Key Case)

- User name.
- Public key.
- Name of issuer which vouches for information.
- Time interval over which data are valid.

RFC 1422 describes the key management architecture for public-key certificates. RFC 1422 and [2] define the certificate format as shown in Figure 2.26.

The integrity of a certificate is checked by verifying the *signature* in the *encrypted* field against the certificate with the public key of the issuer of the certificate.

The authenticity of a certificate is checked by seeing if there is a path leading from the issuer back to the root certificate authority.

2.6.2 Certificate Hierarchy

User certificates are the leaves in a tree with the root certificate authority, the *Internet Policy Registration Authority* (IPRA). The IPRA certifies other certification authorities. These are known as *Policy Certification Authorities* (PCAs). [2] lists three types of PCAs:

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,MIC-ONLY
Content-Domain: RFC822
Originator-Certificate:
 MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFN1Y3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDzAN
 BgNVBAsTBk5PVEFSWTAeFw05MTA5MDQxODM4MTdaFw05MzA5MDMxODM4MTZaMEUx
 CzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGFOYSBTZWN1cml0eSwgSW5jLjEU
 MBIGA1UEAxMLVGVzdCBVc2VyIDEwWTAKBgRVCAEBAgICAANLADBIAkEAwHZHl7i+
 yJcqDtjJCowzTdBJrdAiLAnSC+CnnjOJELyuQiBgkGrgIh3j8/x0fM+YrsyF1u3F
 LZPVtzlndhYFJQIDAQABMAOGCSqGSIb3DQEBAgUAA1kACKr0PqphJYw1j+YPtcIq
 iWlFPuN5jJ79Khfg7ASFxskYkEMjRNZV/HZDZQEhtVaU7Jxfzs2wfX5byMp2X3U/
 5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFN1Y3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDTAL
 BgNVBAsTBFRMQ0EwHhcNOTEwOTAxMDgwMDAwWhcNOTIwOTAxMDc1OTU5WjBRMQsw
 CQYDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4xDzAN
 BgNVBAsTBkJldGEgMTEPMAOGA1UECxMGTk9UQVJZMHAwCgYEVQgBAQICArwDYgAw
 XwJYCsnp6lQCxYykNlODwutF/jMJ3kL+3PjYyHOwk+/9rLg6X65B/LD4bJHtO5XW
 cqAz/7R7XhjYCmOPcqbdzoACZtIlETrKrcJiDYoP+DkZ8k1gCk7hQHpbIwIDAQAB
 MAOGCSqGSIb3DQEBAgUAA38AAICPv4f9Gx/tY4+p+4DB7MV+tKZnvBoy8zgoMGOx
 dD2jMZ/3HsyWKWgSFOeH/AJB3qr9zosG47pyMnTf3aSy2nBO7CMxpUWRBcXUpE+x
 EREZd9++32ofGBIXaialnOgVUn0OzSYgugiQ077nJLDUjOhQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
 jV2OfH+nnXHU8bnL8kPAad/mSQlTDZlbVuxvZAOVRZ5q5+Ejl5bQvqNeqOUNQjr6
 EtE7K2QDeVMCyXsdJlA8fA==

LSBBIG1lc3NhZ2UgZm9yIHVzZSBpbiBOZXNOaW5nLgOKLSBGb2xsb3dpbmcgaXMg
YSBibGFuayBsaW5lOgOKDQpUaGlzIGlzIHRoZSBlbmQuDQo=
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.22 Example MIC-ONLY Message (Public Key Case)

```
To: cert-service@ca.domain
From: requestor@host.domain

-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,CRL-RETRIEVAL-REQUEST
Issuer: <issuer whose latest CRL is to be retrieved>
Issuer: <another issuer whose latest CRL is to be retrieved>
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.23 Example CRL-RETRIEVAL-REQUEST Message

```
To: cert-service@ca.domain
From: requestor@host.domain

-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,CRL
CRL: <CRL to be stored>
Originator-Certificate: <CRL issuer's certificate>
CRL: <another CRL to be stored>
Originator-Certificate: <other CRL issuer's certificate>
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.24 Example CRL Storage Request

```
To: requestor@host.domain
From: cert-service@ca.domain

-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,CRL
CRL: <issuer's latest CRL>
Originator-Certificate: <issuer's certificate>
CRL: <other issuer's latest CRL>
Originator-Certificate: <other issuer's certificate>
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 2.25 Example CRL Retrieval Reply

```
The X.509 certificate format is defined by the following ASN.1
syntax:

Certificate ::= SIGNED SEQUENCE{
        version [0]     Version DEFAULT v1988,
        serialNumber    CertificateSerialNumber,
        signature       AlgorithmIdentifier,
        issuer          Name,
        validity        Validity,
        subject         Name,
        subjectPublicKeyInfo     SubjectPublicKeyInfo,
        issuerUniqueIdentifier  Optional (permitted in version 2 only),
        subjectUniqueIdentifier Optional (permitted in version 2 only),
        algorithmIdentifier     repeat of signature field
        encrypted       signature on all but last of above fields}

Version ::=     INTEGER {v1988(0)}

CertificateSerialNumber ::=     INTEGER

Validity ::=    SEQUENCE{
        notBefore       UTCTime,
        notAfter        UTCTime}

SubjectPublicKeyInfo ::=        SEQUENCE{
        algorithm               AlgorithmIdentifier,
        subjectPublicKey        BIT STRING}


AlgorithmIdentifier ::= SEQUENCE{
        algorithm       OBJECT IDENTIFIER,
        parameters      ANY DEFINED BY algorithm OPTIONAL}

The components of this structure are defined by ASN.1 syntax defined
in the X.500 Series Recommendations.  RFC 1423 provides references
for and the values of AlgorithmIdentifiers used by PEM in the
subjectPublicKeyInfo and the signature data items.  It also describes
how a signature is generated and the results represented.  Because
the certificate is a signed data object, the distinguished encoding
rules (see X.509, section 8.7) must be applied prior to signing.
```

Figure 2.26 Certificate Syntax

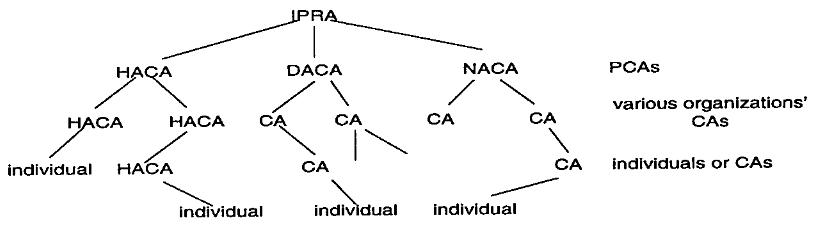

Figure 2.27 PEM Certificate Hierarchy

- High Assurance Certification Authorities (HACAs). HACAs will not grant a certificate to organizations unless they are also highly assured.
- Discretionary Assurance Certification Authorities (DACAs). DACAs do not impose constraints on organizations they certify except to ensure that organizations are who they say they are.
- No Assurance Certification Authorities (NACAs). NACAs have no constraints except they cannot issue two certificates with the same name. No assurance is given that the organizations or people they certify are using their real identities.

Figure 2.27 illustrates the certification tree hierarchy.

2.6.3 Certificate Revocation Lists

A *certificate revocation list* (CRL) is a list of serial numbers of certificates that are invalid, much like a listing of bad credit cards. CRLs are updated periodically, so they also include the period of time they cover.

Figure 2.28 shows the CRL syntax as specified by RFC 1422.

2.7 Integrity in PEM

Integrity is maintained by either *message integrity codes* or *digital signatures.* Both are denoted as MICs in this report. MICs are computed for the message and included as part of the header. In secret key variant, recipients of the message compute the MIC for the message they receive and compare it to the MIC sent in the header. If the MICs match, then the message was unaltered (see Figure 2.8).

```
The following ASN.1 syntax, derived from X.509 and aligned with the
suggested format in recently submitted defect reports, defines the
format of CRLs for use in the PEM environment.

CertificateRevocationList ::= SIGNED SEQUENCE{
        signature       AlgorithmIdentifier,
        issuer          Name,
        lastUpdate      UTCTime,
        nextUpdate      UTCTime,
        revokedCertificates
                        SEQUENCE OF CRLEntry OPTIONAL}

CRLEntry ::= SEQUENCE{
        userCertificate SerialNumber,
        revocationDate UTCTime}
```

Figure 2.28 Certificate Revocation List Syntax

In public key variant, recipients of the message compute the message digest of the message they receive, and verify the MIC sent in the header against the computed message digest with sender's public key. If it succeeds, the message was unaltered (see Figures 2.9 and 2.20).

CRLs and certificates are signed. The signature of a CRL or certificate is included so the recipient can validate the CRL or certificate against the signature which was sent.

2.8 Non-repudiation in PEM

When public-keys are used, signatures provide non-repudiation as only the originator could have created the signature of a message, MIC, CRL, or certificate.

Chapter 3

PEM in Higher-Order Logic

In this chapter, we show the development of all the security functions that are needed to address the security issues raised before: privacy, authentication, integrity, and non-repudiation. The development is done in higher order logic using the HOL system, [5]. Standard predicate calculus notation is used, $\wedge, \vee, \neg, \supset$ denote *and, or, negation*, and *implication.* $\forall$ and $\exists$ denote *for all* and *there exists.* $cond \rightarrow t_1|t_2$ denotes *if cond is true then* t_1 *else* t_2. $\Gamma \vdash t$ denotes a *theorem*, i.e. whenever the list of logical terms in Γ are all true, then the conclusion t is guaranteed to be true. The logical development presented in this paper is a *conservative extension* of the HOL logic, i.e. no axioms were used and the underlying definitions are guaranteed to be consistent. Definitional extensions to HOL are denoted by $\vdash_{def}$.

3.1 Security Functions in HOL

Throughout this report, we identify a person by his/her keys. In public key cryptography, the person is identified by public key which is known to everyone. Since a private key belongs to only one owner, the corresponding public key uniquely identifies a person. In secret key cryptography, two or more people who share a secret key are identified by that secret; a key uniquely identifies the group who shares it.

3.1.1 Privacy

Function **is_Private** checks the privacy property of a mail message. It declares the message as private if the decrypted received message matches that of the original plaintext.

Since both secret key encryption and public key encryption are used to protect the privacy of messages, two variants of **is_Private** are given. The difference between **is_PrivateS** for secret key and **is_PrivateP** for public key is: secret key encryption takes an initial vector while public key encryption does not.

is_PrivateS has parameters: 1) *decryptS* - a secret key decryption function, 2) *message* - the original plaintext, 3) *rxmsg* - the received (encrypted) message, 4) *decryptIV* - initial vector for decryption and 5) *key* - the shared secret key.

```
is_PrivateS
   ⊢def  ∀decryptS message rxmsg decryptIV key.
         is_PrivateS decryptS message rxmsg decryptIV key =
         decryptS rxmsg key decryptIV = message
```

is_PrivateP has parameters: 1) *decryptP* - a public key decryption function, 2) *message* - the original plaintext, 3) *rxmsg* - the received ciphertext and 4) *dkey* - the private key of the recipient

```
is_PrivateP
   ⊢def  ∀decryptP message rxmsg dkey.
         is_PrivateP decryptP message rxmsg dkey =
         decryptP rxmsg dkey = message
```

is_Private is true if and only if there is one and only one person who can read the original message, namely the intended recipient.

When a mail message satisfies assumptions listed below, the correctness theorem of **is_Private** can be proved by using definitions of **is_PrivateS** and **is_PrivateP**. The assumptions are: 1) The received message is the same as the transmitted message, 2) the transmitted message is the original message encrypted with a key (either a shared secret key, or a public key), 3) for any encryption key, (in either secret key cryptography or public key cryptography), there is an unique decryption key which can be used to retrieve the original text. They are taken as antecedents of a nested implication.

Theorem **is_Private_DEK** is the privacy property of the DEK used in PEM which is encrypted with the recipient's public key and is retrieved using the recipient's private key. (See Figure 2.18.)

```
is_Private_DEK
   ⊢ ∀decryptP encryptP message txmsg rxmsg ekey dKEY0 dkey.
        (rxmsg = txmsg) ⊃
        (txmsg = encryptP message ekey) ⊃
        (∀msg. decryptP (encryptP msg ekey) dKEY0 = msg) ⊃
        (∀msg d2.
          (decryptP (encryptP msg ekey) d2 = msg) ⊃  (d2 = dKEY0)) ⊃
        ((dkey = dKEY0) = is_PrivateP decryptP message rxmsg dkey)
```

Theorem **is_Private_msg** is the privacy property of the original plaintext message in PEM which is retrieved with the DEK. Since DEK is known only to the intended recipient, as proved by theorem **is_Private_DEK**, the confidentiality of the message is preserved.

```
is_Private_msg
   ⊢ ∀decryptS encryptS message txmsg rxmsg decryptIV KEY0 key.
        (rxmsg = txmsg) ⊃
        (txmsg = encryptS message KEY0 decryptIV) ⊃
        (∀msg key.
          (decryptS (encryptS msg key decryptIV) key decryptIV = msg) ∧
          (∀msg key1. (decryptS msg key1 decryptIV
             = decryptS msg key decryptIV) = key = key1)) ⊃
        ((key = KEY0) =
        is_PrivateS decryptS message rxmsg decryptIV key)
```

In both cases, if the received message is not the same as that transmitted, that is, either the data exchange key (DEK) is modified or the encrypted message is modified over the net, the intended recipient of the message will not be able to read it. The plaintext message is still private since nobody else can retrieve it, but the recipient encounters a denial-of-service attack here.

3.1.2 Source Authentication

We have defined source authentication in two ways. If verification of the signature against the received message succeeds, the recipient is sure of the source of the received message. In **is_Authentic**, the signature is verified against the original message. In **is_Authentic2**, the MIC (digital signature) of the message is verified against the hash of a message.

The parameters **is_Authentic** takes are: 1) *verify* - public key signature verification function, 2) *message* - plaintext, 3) *signature* - signature of the plaintext, 4) *ekey* - signer's public key.

```
is_Authentic
    ⊢def  ∀verify message signature ekey.
          is_Authentic verify message signature ekey =
          verify message signature ekey
```

The parameters **is_Authentic2** takes are: 1) *verify* - public key signature verification function, 2) *hash* - message digest algorithm, 3) *message* - plaintext, 4) *mic* - digital signature of the plaintext, 5) *ekey* - signer's public key.

```
is_Authentic2
    ⊢def  ∀verify hash message mic ekey.
          is_Authentic2 verify hash message mic ekey =
          verify (hash message) mic ekey
```

The desired property of source authentication is the check is true if and only if the originator of the message is the one identified by the public key we use to verify the signature.

The assumptions we made on source authentication are: 1) the received message is the same as transmitted, 2) the transmitted message is a digital signature of plaintext, 3) there is an unique private key *dKEY0* associated with a signature which can be verified through the corresponding public key *ekey*.

In the following theorem it is proved that if these assumptions are satisfied, the originator of the transmitted plaintext is known if and only if it passes the **is_Authentic2** check.

```
is_Authentic_msg
    ⊢  ∀verify sign hash message txmic rxmic ekey dKEY0 dkey.
          (rxmic = txmic) ⊃
          (txmic = sign (hash message) dkey) ⊃
          (∀m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey = dkey2 = dKEY0)⊃
          ((dkey = dKEY0) = is_Authentic2 verify hash message rxmic ekey)
```

If the first assumption is not satisfied, the source authentication fails and and the recipient of the message cannot be sure of the source of the message.

```
not_Authentic
    ⊢ ∀verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
        (txmic = sign (hash MESSAGE0) dKEY0) ⊃
        (∀m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ⊃
        (∀m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
            ⊃  (m1 = m2) ∧ (dkey1 = dkey2)) ⊃
        ¬(rxmic = txmic) ⊃
        ¬(is_Authentic2 verify hash MESSAGE0 rxmic ekey)
```

3.1.3 Integrity

is_Intact is defined for message integrity checking. It takes several parameters: 1)*verify* - a function verifies the signature, which takes a plaintext message, a signature and a key, and returns *true* if the signature is signed on the given plaintext with the private key paired with the given key, otherwise, it returns *false.* 2) *hash* - the message digest algorithm; 3) *message* - the plaintext part of the message retrieved from the mail; 4) *ekey* - the public key of originator used by the recipient to verify a signature; and 5) *mic* - the received digital signature of the message.

It declares both the message and its digital signature are intact if the verification of the digital signature of the original message against the hash of the received message succeeds. The definition matches the scheme shown in Figure 2.9.

```
is_Intact
    ⊢def  ∀verify hash message mic ekey.
          is_Intact verify hash message mic ekey =
          verify (hash message) mic ekey
```

The assumptions made about the received message are: 1) the received signature is generated by signing the hash (message digest) of the transmitted message. 2) it is computationally infeasible to find two messages m_1 and m_2 which hash to the same value, so if two hashes are equal the two messages are the same; 3) the verification process succeeds if and only if the signature is generated on the plaintext that is being verified.

What we want is for **is_Intact** to be true is-and-only-if the received message is identical to the one transmitted. Under these assumptions, the correctness theorem is proved using the definition of **is_Intact** with the assumed properties in the antecedent of the nested implication.

```
is_Intact_msg
   ⊢  ∀verify sign hash txmessage rxmessage txmic rxmic ekey dkey.
        (txmic = sign (hash txmessage) dkey) ⊃
        (rxmic = txmic) ⊃
        (∀m1 m2. (hash m1 = hash m2) ⊃  (m1 = m2)) ⊃
        (∀s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ⊃
        ((rxmessage = txmessage)
          = is_Intact verify hash rxmessage rxmic ekey)
```

When the received MIC is not the same as the one sent by originator, the following theorem proves that the recipient cannot be sure the integrity of either MIC or plaintext message.

```
not_Intact =
   ⊢  ∀verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
        (txmic = sign (hash MESSAGE0) dKEY0) ⊃
        (∀m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ⊃
        (∀m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
            ⊃  (m1 = m2) ∧ (dkey1 = dkey2)) ⊃
        ¬(rxmic = txmic) ⊃
        ¬(is_Intact verify hash MESSAGE0 rxmic ekey)
```

3.1.4 Non-Repudiation

is_non_Deniable is the security check of the non-repudiation of the message system. It checks the non-deniability of the sender of the message by verifying the signature against the received plaintext. It has following parameters: 1) *verify* - public key signature verification function, 2) *message* - original plaintext, 3) *signature* - signature of the plaintext, 4) *ekey* - signer's public key. Since both source-authentication and non-repudiation of a message is obtained through its signature, **is_non_Deniable** is defined in the same way as **is_Authentic**.

```
is_non_Deniable
   ⊢def  ∀verify message signature ekey.
         is_non_Deniable verify message signature ekey =
         verify message signature ekey
```

The assumptions we made for checking the non-deniability of a message are: 1) the received MIC is the same as the transmitted MIC, 2) the transmitted MIC is generated by the originator on plaintext MESSAGE0, 3) it is computationally infeasible to find two messages *m1* and *m2* which hash to the same value, so if two hashes are equal the two messages are the same. 4) it is computationally infeasible to find two messages *m1* and *m2* and two private keys *k1* and *k2*, which can generate same signature, so if we can verify one signature against a message with a public key, then the private key and the plaintext used to generate signature are unique. If the above assumptions are satisfied, the verification process succeeds if and only if the signature is generated on the plaintext that is being verified with the unique private key that is known only to the signer. This scheme matches that shown in Figure 2.9.

Under the above assumptions, the non-repudiation check is true if and only if the received message is generated by the originator whose public key is *ekey*, so that the originator cannot deny having sent the message. The correctness theorem **is_non_Deniable_msg** is proved using the definition of **is_non_Deniable**.

```
is_non_Deniable_msg
    ⊢ ∀verify sign hash message MESSAGE0 txmic rxmic ekey dKEY0 dkey.
          (rxmic = txmic) ⊃
          (txmic = sign (hash MESSAGE0) dkey) ⊃
          (∀m1 m2. (hash m1 = hash m2) = m1 = m2) ⊃
          (∀m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
             = (m1 = m2) ∧ (dkey2 = dKEY0)) ⊃
          ((dkey = dKEY0) ∧ (message = MESSAGE0) =
          is_non_Deniable verify (hash message) rxmic ekey)
```

When the received MIC is not the same as transmitted MIC, then the recipient cannot show to a third party that the originator has indeed sent the message. This is shown in the theorem follows.

```
is_deniable =
    ⊢ ∀verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
          (txmic = sign (hash MESSAGE0) dKEY0) ⊃
          (∀m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ⊃
          (∀m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
              ⊃  (m1 = m2) ∧ (dkey1 = dkey2)) ⊃
          ¬(rxmic = txmic) ⊃
          ¬(is_non_deniable verify (hash MESSAGE0) rxmic ekey)
```

The definitions and properties developed in this section are independent of any particular implementation. What we must do is link the particular implementation to the general definitions and properties. For this we must define the structure of PEM messages in detail.

3.2 Message Structure in HOL

Each PEM message type has public key variant and private key variant. In this section, only the public key variant will be discussed, since it is the only one in use. Also, some PEM messages are encoded to avoid the "mailer mangling" problem. Encoding is not discussed here as it does not contribute to the security services we are concerned with in this report.

As an example, we discuss the structure of MIC-CLEAR messages using public-key signature algorithms. Table 2.4 shows the format of MIC-ONLY and MIC-CLEAR messages using public keys. Figure 2.22 is an example of a MIC-ONLY message. MIC-ONLY messages encode their messages to avoid mailer problems. MIC-CLEAR messages do not use encoding.

MIC-CLEAR messages are *8-tuples*: $(preeb \times proctype \times contentdomain \times id\_asymmetric \times (certificate)list \times MIC\_info \times pemtext \times posteb)$ as shown in Table 2.4. However, not all *8-tuples* are valid MIC-CLEAR messages. When a proper subset of possible representations is identified as a *new type*, reasoning about messages is simplified because only valid representations are considered. The next section briefly illustrates the concepts of defining new types in HOL.

3.2.1 Type Definition in HOL

New types are introduced in HOL by identifying a subset of an existing type whose properties correspond to the properties of the new type, [11]. *Isomorphic* (one-to-one and onto) mappings between elements of the new type and elements of the subset of the existing type are defined. One mapping is the *representation* of the new type in terms of the existing type. The other is the *abstraction* of the existing type into the new type.

For example, say we wish to introduce the type *color* which has only two members, *black* and *white*. In BNF, we write:

$$color ::= black \mid white \tag{3.1}$$

Suppose we choose to represent *color* by the cartesian product $bool \times bool$. There are four elements in $bool \times bool$ but only two are needed. We choose to represent *black* as (T, F) and *white* as (F, T) as shown in Figure 3.1.

Defining new types in HOL is a three-step process. The first step finds an appropriate subset of an existing type to represent the new type. The second step extends the syntax of HOL to include the new type by using

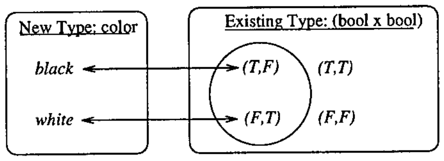

Figure 3.1 Defining Type *color*

a *type definition axiom* which defines the relationship between the new type and its representation. Finally, from the type definition axiom, the properties of the new type are derived.

In our example, the valid representation of boolean pairs is defined by *is_Color*.

$$is\_Color\ (x,y) = ((x,y) = (T,F) \vee (x,y) = (F,T)) \tag{3.2}$$

As there is at least one value of (x,y) which satisfies *is_Color*, the following type definition axiom holds which states that there is a representation function *rep* which is isomorphic between *black* and *white* and (T,F) and (F,T).

$$\begin{aligned} &\vdash \exists\, rep : color \rightarrow (bool \times bool). \\ &\quad (\forall a_1\ a_2 . rep\ a_1 = rep\ a_2 \supset a_1 = a_2) \wedge \\ &\quad (\forall r : (bool \times bool).is\_Color\ r = (\exists a : color.r = rep\ a)) \end{aligned} \tag{3.3}$$

A valid representation function for *color* is *any* function which has the isomorphic properties defined above.

We refer to objects having a property P with Hilbert's ε-operator, [11]. The semantics of ε are given below.

$$\vdash \forall P.(\exists x.P\ x) \supset P(\varepsilon x.P\ x) \tag{3.4}$$

For example, if $P\ x$ were $1 < x < 4$ where x is a natural number, $\varepsilon x.P\ x$ would be either 2 or 3 and $P(\varepsilon x.P\ x)$ is *true*.

We define the representation and abstraction functions *REP_color* and *ABS_color* as follows.

$$\begin{aligned} &TYPE\_DEF\ P\ rep = \\ &\quad (\forall a_1\ a_2.rep\ a_1 = rep\ a_2 \supset a_1 = a_2) \land \\ &\quad (\forall r.P\ r = (\exists a : color.r = rep\ a)) \end{aligned} \tag{3.5}$$

$$REP\_color = \varepsilon rep.TYPE\_DEF\ is\_Color\ rep \tag{3.6}$$
$$ABS\_color\ r = (\varepsilon a.r = REP\_color\ a) \tag{3.7}$$

REP_color is *any* function satisfying the one-to-one and onto properties of *TYPE_DEF*. *ABS_color r* returns a color whose representation is r. Given the associations in Figure 3.1 we define *black* and *white* as follows.

$$black = ABS\_color\ (T,F) \tag{3.8}$$
$$white = ABS\_color\ (F,T) \tag{3.9}$$

Given the definitions of *TYPE_DEF, REP_color,* the semantics of ε, and *ABS_color*, the following properties are easily proved. These properties state that *REP_color* is one-to-one and onto; *ABS_color* is one-to-one and onto within the constraints of *is_Color*; and *REP_color* and *ABS_color* invert each other.

$$\begin{aligned} &\vdash \forall a_1\ a_2. \\ &\quad REP\_color\ a_1 = REP\_color\ a_2 \supset (a_1 = a_2) \end{aligned} \tag{3.10}$$
$$\vdash \forall r.is\_Color\ r = (\exists a.r = REP\_color\ a) \tag{3.11}$$
$$\begin{aligned} &\vdash \forall r_1\ r_2.is\_Color\ r_1 \supset (is\_Color\ r_2 \supset \\ &\quad (ABS\_color\ r_1 = ABS\_color\ r_2 \supset r_1 = r_2)) \end{aligned} \tag{3.12}$$
$$\vdash \forall a.\exists r.(a = ABS\_r) \land is\_Color\ r \tag{3.13}$$
$$\vdash \forall a.ABS\_color(REP\_color\ a) = a \tag{3.14}$$
$$\vdash \forall r.is\_Color\ r = (REP\_color(ABS\_color\ r) = r) \tag{3.15}$$

The same techniques used to define *color* are generally applicable. In the next section, we show how to apply type definition techniques to the message integrity code field of messages.

3.2.2 MIC_info as a Type

We focus on the *MIC_info* portion of a message. Figure 2.13 gives the BNF definition of `<micinfo>`. It is a *3-tuple* where the first element identifies the hash function used to compute the MIC; the second element is the signature algorithm used to encrypt the MIC; and the third element is the signed message digest for the transmitted message. The particular algorithms are defined in Figure 2.15.

As some of the algorithms (like RSA) are used in more than one capacity, we first introduce the algorithm identifiers as a separate *abstract* type – *algid*, i.e. we do not care about how the members of the type are actually represented.

$$\begin{aligned} algid ::= {} & DES\_CBC|DES\_EDE|DES\_ECB| \\ & RSA|RSA\_MD2|RSA\_MD5 \end{aligned} \tag{3.16}$$

Valid *MIC_Info* fields are a *proper* subset of all *3-tuples* of $(algid \times algid \times asymsignmic)$ The predicate *is_MIC_info* identifies the valid *3-tuples* for *MIC_Info*. Note, *FST* and *SND* are destructors for pairs, e.g. $FST\ (a,b,c) = a$ and $SND\ (a,b,c) = (b,c)$.

$$\begin{aligned} & is\_MIC\_info\ x = \\ & \quad ((FST\ x = RSA\_MD2) \lor \\ & \quad\ (FST\ x = RSA\_MD5)) \land \\ & \quad ((FST(SND\ x) = DES\_EDE) \lor \\ & \quad\ (FST(SND\ x) = DES\_ECB) \lor \\ & \quad\ (FST(SND\ x) = RSA)) \end{aligned} \tag{3.17}$$

From the definition of *is_MIC_info* we can prove the theorem $\vdash \exists x.is\_MIC\_info\ x$ which allows us to introduce a new type *MIC_info* as follows.

$$\vdash \exists rep.TYPE\_DEF\ is\_MIC\_info\ rep \tag{3.18}$$

Using the above type definition axiom, we define the representation function *REP_MIC_info* and the abstraction function *MIC_info* as follows, (notice that the abstraction function is the same name as the MIC-info field identifier).

$$REP\_MIC\_info = \\ \quad \varepsilon rep.TYPE\_DEF\ is\_MIC\_info\ rep \tag{3.19}$$

$$MIC\_info\ r = (\varepsilon a.r = REP\_MIC\_info\ a) \tag{3.20}$$

The properties of REP_MIC_info and MIC_info are proved in exactly the same way as the properties of the representation and abstraction functions are for *color*. The next two theorems state that REP_MIC_info is one-to-one and onto.

$$\vdash \forall a\ a'.(REP\_MIC\_info\ a = REP\_MIC\_info\ a') \supset a = a' \tag{3.21}$$

$$\vdash \forall r.is\_MIC\_info\ r = (\exists a.r = REP\_MIC\_info\ a) \tag{3.22}$$

The next two theorems state that MIC_info is one-to-one and onto.

$$\vdash \forall r\ r'.is\_MIC\_info\ r \supset is\_MIC\_info\ r' \supset \\ \quad ((MIC\_Info\ r = MIC\_Info\ r') \supset r = r') \tag{3.23}$$

$$\vdash \forall a.\exists r.(a = MIC\_Info\ r) \land is\_MIC\_info\ r \tag{3.24}$$

The next two theorems state that MIC_info and REP_MIC_info are inverses.

$$\vdash \forall a.MIC\_Info(REP\_MIC\_info\ a) = a \tag{3.25}$$

$$\vdash \forall r.is\_MIC\_info\ r = REP\_MIC\_info(MIC\_Info\ r) = r \tag{3.26}$$

Since MIC-CLEAR messages are *8-tuples*, and given the formal definition of each of the message fields as data types, we define various accessor functions to get the MIC hash, signature, and signed MIC portions of the *MIC_Info* field.

$$get\_MIC\_algid\ x = FST(REP\_MIC\_info\ x) \tag{3.27}$$

$$get\_MIC\_sigalgid\ x = FST(SND(REP\_MIC\_info\ x)) \tag{3.28}$$

$$get\_MIC\_mic\ x = SND(SND(REP\_MIC\_info\ x)) \tag{3.29}$$

Based on the above characterization of *MIC_info* as a type, any x which is a member of *MIC_info* has a valid representation as a *3-tuple* (*algid* $\times$ *algid* $\times$ *asymsignmic*). This is stated by the following theorem.

$$\vdash \forall x.is\_MIC\_info(REP\_MIC\_info\ x) \tag{3.30}$$

The above theorem coupled with the definition of *is_MIC_info* leads to the following correctness properties of the hash and signature accessor functions. In particular, each accessor function when applied to a valid *MIC_info* field will yield only the specified hash and signature algorithms.

$$\begin{aligned} \vdash \forall x.&(get\_MIC\_algid\ x = RSA\_MD2)\lor \\ &(get\_MIC\_algid\ x = RSA\_MD5) \end{aligned} \tag{3.31}$$

$$\begin{aligned} \vdash \forall x.&(get\_MIC\_sigalgid\ x = DES\_EDE)\lor \\ &(get\_MIC\_sigalgid\ x = DES\_ECB)\lor \\ &(get\_MIC\_sigalgid\ x = RSA) \end{aligned} \tag{3.32}$$

As the algorithm names in the *MIC_info* field are just names and not the actual hash and signature functions, we define signature and hash selector functions which take a function name and return its corresponding function. For simplicity, we do not define the actual functions here, but just define them as function names with the proper type signatures. For example, *fDES_EDE* is of type *asymsignmic* $\rightarrow$ *key* $\rightarrow$ *asymsignmic* and is the signature function corresponding to DES_EDE.

$$\begin{aligned} &MIC\_sign\_select\ x = \\ &\quad ((get\_MIC\_sigalgid\ x = DES\_EDE) \rightarrow sDES\_EDE \\ &\quad\ |((get\_MIC\_sigalgid\ x = DES\_ECB) \rightarrow sDES\_ECB|sRSA)) \end{aligned} \tag{3.33}$$

$$\begin{aligned} &MIC\_hash\_select\ x = \\ &\quad ((get\_MIC\_algid\ x = RSA\_MD2) \rightarrow fRSA\_MD2|fRSA\_MD5) \end{aligned} \tag{3.34}$$

Other selector and accessor functions are defined similarly and have properties similar to those shown above. The development of these functions is listed in the appendices.

3.3 Functions for MIC-CLEAR Messages

Given the MIC accessor functions for MIC-CLEAR message, the hash and signature selector functions, and the general integrity checking function **is_Intact**, we now define the integrity checking function **MIC_CLEAR_is_-Intact** for MIC-CLEAR messages. It is the general integrity function **is_Intact** with its parameters specialized with the hash and signature selection functions.

```
MIC_CLEAR_is_Intact msg =                                          (3.35)
    (let micInfo = get_MIC_CLEAR_MIC_Info msg
    in
     let ekey =
      get_Key_from_ID (get_OriginatorAsymID_info msg)
    in
     is_Intact
      (MIC_sign_select micInfo)
       (MIC_hash_select micInfo)
        (get_MIC_CLEAR_text msg)
         (get_MIC_mic micInfo) ekey)
```

Given the definition of **MIC_CLEAR_is_Intact** and the general correctness theorem **is_Intact**, we can prove the following correctness theorem for **MIC_CLEAR_is_Intact**. It states that under similar assumptions to the general *is_Intact* correctness theorem, **MIC_CLEAR_is_Intact** is true if-and-only-if the transmitted and received messages are the same. When **MIC_CLEAR_is_Intact** is false, then what was received differs from what was transmitted. The theorem assures that given the assumptions the intent of the integrity function is satisfied for MIC-CLEAR messages. Similar functions for other message types and security properties can be defined and verified.

```
⊢ ∀mic_clear_msg sign txmessage dkey.                              (3.36)
    let micInfo =
     get_MIC_CLEAR_MIC_Info mic_clear_msg
    in
    let ekey = get_Key_from_ID
      (get_Originator_AsymID_info mic_clear_msg)
    in
    let hash = MIC_hash_select micInfo
```

and
$verify = MIC\_sign\_select\ micInfo$
and
$rxmessage = get\_MIC\_CLEAR\_text\ mic\_clear\_msg$
in
$(get\_MIC\_mic\ micInfo = sign\ (hash\ txmessage)\ dkey)$
$\supset$
$(\forall m1\ m2.(hash\ m1 = hash\ m2) \supset (m1 = m2)) \supset$
$(\forall m1\ m2.verify\ m1\ (sign\ m2\ dkey)\ ekey = m1 = m2) \supset$
$((txmessage = rxmessage) =$
$MIC\_CLEAR\_is\_Intact\ mic\_clear\_msg)$

3.4 Functions for ENCRYPTED Messages

For simplicity, *ENCRYPTED* messages are modeled as an *8-tuple*: $(preeb \times proctype \times contentdomain \times dekinfo \times id\_asymmetric \times (certificate)list \times MIC\_info \times (id\_asymmetric \times Key\_info)list \times pemtext \times posteb)$.

The security, accessor, and selector functions for ENCRYPTED messages are defined in the same way as they are for MIC-CLEAR messages. They are the general security functions with the parameters specialized with the selection functions. We assume all the fields in PEM message are successfully retrieved, except the ciphertext and encrypted MIC fields.

With the specialized security functions and the general correctness theorems, we can prove the specialized correctness theorems for *ENCRYPTED* messages.

3.4.1 Privacy

The privacy check functions **ENCRYPTED_is_PrivateP** and **ENCRYPTED_is_PrivateS** are defined using the general privacy functions **is_PrivateP** and **is_PrivateS** with their parameters specialized with hash and signature selection functions.

```
ENCRYPTED_is_PrivateP
    ⊢def  ∀msg txDEK.
          ENCRYPTED_is_PrivateP msg txDEK =
          is_PrivateP (DEK_encrypt_select (getEN_KEY_info msg)) txDEK
            (getEN_msg_EncryptedKey msg) recipientkey
```

```
ENCRYPTED_is_PrivateS
    ⊢def  ∀msg message.
          ENCRYPTED_is_PrivateS msg message =
          (let rxDEK = getEN_msg_DEK msg
           and
           decryptIV = getEN_msg_MsgEncryptIV msg
           in
           is_PrivateS (msg_Encrypt_select (getEN_DEK_info msg)) message
             (getEN_Message_info msg) decryptIV rxDEK)
```

Given the definitions of **ENCRYPTED_is_PrivateP** and **ENCRYPTED_is_PrivateS** and the general correctness theorems **is_Private_DEK** and **is_Private_msg**, we can prove the correctness theorems for **ENCRYPTED_is_Private_DEK** and **ENCRYPTED_is_Private_msg**. The following theorem states that under similar assumptions to the general **is_Private_DEK** correctness theorem, **ENCRYPTED_is_Private_DEK** is true if-and-only-if the received DEK is not disclosed during transmission.

```
ENCRYPTED_is_Private_DEK
    ⊢  ∀Encrypted_msg encryptP DEK dKEYO dkey.
          let Key_info = getEN_KEY_info Encrypted_msg
          in
          let decryptP = DEK_encrypt_select Key_info
          and
          rxmsg = getEN_msg_EncryptedKey Encrypted_msg
          and
          dkey = recipientkey
          in
          (rxmsg = txmsg) ⊃
          (txmsg = encryptP DEK ekey) ⊃
          (∀msg. decryptP (encryptP msg ekey) dKEYO = msg) ⊃
          (∀msg d2.
            (decryptP (encryptP msg ekey) d2 = msg) ⊃  (d2 = dKEYO)) ⊃
          ((dkey = dKEYO) = ENCRYPTED_is_PrivateP Encrypted_msg DEK)
```

The theorem below states that under similar assumptions to the general **is_Private_msg** correctness theorem, **ENCRYPTED_is_Private_msg** is true if-and-only-if the received original plaintext is not disclosed during transmission.

```
ENCRYPTED_is_Private_msg
    ⊢  ∀Encrypted_msg encryptS message DEK.
          let DEK_info = getEN_DEK_info Encrypted_msg
          in
          let decryptS = msg_Encrypt_select DEK_info
          and
          rxmsg = getEN_Message_info Encrypted_msg
          and
          decryptIV = getEN_msg_MsgEncryptIV Encrypted_msg
          and
          KEY0 = DEK
          and
          key = getEN_msg_DEK Encrypted_msg
          in
          (rxmsg = txmsg) ⊃
          (txmsg = encryptS message KEY0 decryptIV) ⊃
          (∀msg key.
            (decryptS (encryptS msg key decryptIV) key decryptIV = msg) ∧
            (∀msg key1.
              (decryptS msg key1 decryptIV = decryptS msg key decryptIV) =
              key = key1)) ⊃
          ((key = KEY0) = ENCRYPTED_is_PrivateS Encrypted_msg message)
```

3.4.2 Source Authentication

The source authentication check function **ENCRYPTED_is_Authentic2** is defined as the general source authentication function **is_Authentic2** with its parameters specialized with the hash and signature selection functions.

```
ENCRYPTED_is_Authentic2
    ⊢def  ∀msg.
          ENCRYPTED_is_Authentic2 msg =
          (let micInfo = getEN_MIC_info msg
           in
           let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
           in
           is_Authentic2 (MIC_sign_select micInfo)
             (MIC_hash_select micInfo) (getEN_msg_message msg)
             (getEN_msg_MIC msg) ekey)
```

Given the definition of **ENCRYPTED_is_Authentic2** and the general correctness theorem **is_Authentic_msg**, we prove the correctness theorem for **ENCRYPTED_is_Authentic_msg**. It states that under similar assumptions to the general **is_Authentic_msg** correctness theorem, **ENCRYPTED_is_Authentic_msg** is true if-and-only-if the received original plaintext is sent by the originator identified by the public key stated in the received message.

```
ENCRYPTED_is_Authentic_msg
    ⊢  ∀Encrypted_msg sign txmic dKEY0 dkey.
          let micInfo = getEN_MIC_info Encrypted_msg
          in
          let verify = MIC_sign_select micInfo
          and
          hash = MIC_hash_select micInfo
          and
          message = getEN_msg_message Encrypted_msg
          and
          rxmic = getEN_msg_MIC Encrypted_msg
          and
          ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
          in
          (rxmic = txmic) ⊃
          (txmic = sign (hash message) dkey) ⊃
          (∀m1 m2 dkey2.
             verify m1 (sign m2 dkey2) ekey = dkey2 = dKEY0) ⊃
          ((dkey = dKEY0) = ENCRYPTED_is_Authentic2 Encrypted_msg)
```

3.4.3 Integrity

The integrity check function **ENCRYPTED_is_Intact** is defined as the general integrity function **is_Intact** with its parameters specialized with the hash and signature selection functions.

```
ENCRYPTED_is_Intact
    ⊢def  ∀msg.
          ENCRYPTED_is_Intact msg =
          (let micInfo = getEN_MIC_info msg
           in
           let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
           in
           is_Intact (MIC_sign_select micInfo) (MIC_hash_select micInfo)
             (getEN_msg_message msg)
             (getEN_msg_MIC msg) ekey)
```

Given the definition of **ENCRYPTED_is_Intact** and the general correctness theorem **is_Intact_msg**, the correctness theorem **ENCRYPTED_-is_Intact_msg** can be proved. The theorem states that under similar assumptions to the general **is_Intact_msg** correctness theorem, **ENCRYPTED_is_Intact_msg** is true if-and-only-if the received plaintext after processing is the same as the original plaintext before encryption.

```
ENCRYPTED_is_Intact_msg
    ⊢  ∀Encrypted_msg sign txmessage txmic dkey.
         let micInfo = getEN_MIC_info Encrypted_msg
         in
         let verify = MIC_sign_select micInfo
         and
         hash = MIC_hash_select micInfo
         and
         rxmessage = getEN_msg_message Encrypted_msg
         and
         rxmic = getEN_msg_MIC Encrypted_msg
         and
         ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
         in
         (txmic = sign (hash txmessage) dkey) ⊃
         (rxmic = txmic) ⊃
         (∀m1 m2. (hash m1 = hash m2) ⊃  (m1 = m2)) ⊃
         (∀s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ⊃
         ((rxmessage = txmessage) = ENCRYPTED_is_Intact Encrypted_msg)
```

3.4.4 Non-Repudiation

The non-repudiation check function **ENCRYPTED_is_non_deniable** is defined as the general non-deniability function **is_non_deniable** with specialized parameters for hash and signature selection functions.

```
ENCRYPTED_is_non_deniable
    ⊢def  ∀msg.
          ENCRYPTED_is_non_deniable msg =
          (let micInfo = getEN_MIC_info msg
           in
           let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
           and
           hash = MIC_hash_select micInfo
           in
           is_non_deniable (MIC_sign_select micInfo)
             (hash (getEN_msg_message msg))
             (getEN_msg_MIC msg) ekey)
```

Given the definitions of **ENCRYPTED_is_non_deniable** and the general correctness theorem **is_non_deniable_msg**, we can prove the correctness theorem for **ENCRYPTED_is_non_deniable_msg**. It states that under similar assumptions to the general **is_non_deniable_msg** correctness theorem, **ENCRYPTED_is_non_deniable_msg** is true if-and-only-if the originator of the retrieved plaintext identified by the public key stated in the received message cannot deny having sent the message.

```
ENCRYPTED_is_non_deniable_msg
   ⊢ ∀Encrypted_msg sign MESSAGE0 txmic dKEY0 dkey.
        let micInfo = getEN_MIC_info Encrypted_msg
        in
        let verify = MIC_sign_select micInfo
        and
        hash = MIC_hash_select micInfo
        and
        message = getEN_msg_message Encrypted_msg
        and
        rxmic = getEN_msg_MIC Encrypted_msg
        and
        ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ⊃
        (txmic = sign (hash MESSAGE0) dkey) ⊃
        (∀m1 m2. (hash m1 = hash m2) = m1 = m2) ⊃
        (∀m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey =
           (m1 = m2) ∧ (dkey2 = dKEY0)) ⊃
        ((dkey = dKEY0) ∧ (message = MESSAGE0) =
        ENCRYPTED_is_non_deniable Encrypted_msg)
```

Chapter 4

Conclusions

The increased use of networked and distributed computing makes security a major concern. The capability to verify that a system meets its security requirements will continue to grow in importance. In particular, the capability to assign security properties to engineering structures is crucial.

This work focuses on verifying the security properties of Privacy Enhanced Mail (PEM). Security properties such as privacy, source authentication, integrity and non-repudiation are defined independently of any implementation structure. PEM message structures and operations on those structures are shown to have the desired security properties. Various PEM structures are defined as types. Security interpretations are defined as operations on these types.

All the definitions and proofs are done using the Higher Order Logic (HOL) theorem-prover. While at times the proofs are intricate, the proofs are well within the capabilities of engineers who have been trained to use HOL.

The work done on PEM shows the feasibility of using formal logic and computer assisted reasoning tools to describe and verify relatively complex systems. The advantages of using these methods is the assurance of correctness of the specifications given to implementers. If the specifications are correctly implemented, then the desired security properties will be achieved.

Appendix A

NOTATIONAL CONVENTIONS

This appendix is excerpted in part from RFC 822, *Standard for the Format of ARPA Internet Text Messages*, [3]. It defines the augmented BNF notation used for describing PEM message formats.

This specification uses an augmented Backus-Naur Form (BNF) notation. The differences from standard BNF involve naming rules and indicating repetition and "local" alternatives.

A.1 RULE NAMING

Angle brackets ("<", ">") are not used, in general. The name of a rule is simply the name itself, rather than "<name>". Quotation-marks enclose literal text (which may be upper and/or lower case). Certain basic rules are in uppercase, such as SPACE, TAB, CRLF, DIGIT, ALPHA, etc. Angle brackets are used in rule definitions, and in the rest of this document, whenever their presence will facilitate discerning the use of rule names.

A.2 RULE1 / RULE2: ALTERNATIVES

Elements separated by slash ("/") are alternatives. Therefore "foo / bar" will accept foo or bar.

A.3 (RULE1 RULE2): LOCAL ALTERNATIVES

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo / bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

A.4 *RULE: REPETITION

The character "*" preceding an element indicates repetition. The full form is:

<l>*<m>element

indicating at least <l> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

A.5 [RULE]: OPTIONAL

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

A.6 NRULE: SPECIFIC REPETITION

"<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

A.7 #RULE: LISTS

A construct "#" is defined, similar to "*", as follows:

<l>#<m>element

indicating at least <l> and at most <m> elements, each separated by one or more commas (","). This makes the usual form of lists very easy; a rule such as '(element *("," element))' can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element),,(element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element must be present. Default values are 0 and infinity so that "#(element)" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

A.8 ; COMMENTS

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

A.9 ALPHABETICAL LISTING OF SYNTAX RULES

```
address     =  mailbox                         ; one addressee
            /  group                           ; named list
addr-spec   =  local-part "@" domain           ; global address
ALPHA       =  <any ASCII alphabetic character>
                                               ; (101-132, 65.- 90.)
                                               ; (141-172, 97.-122.)
atom        =  1*<any CHAR except specials, SPACE and CTLs>
authentic   =   "From"       ":"   mailbox  ; Single author
            / ( "Sender"     ":"   mailbox  ; Actual submittor
                "From"       ":" 1#mailbox) ; Multiple authors
                                            ;  or not sender
CHAR        =  <any ASCII character>        ; (  0-177,  0.-127.)
comment     =  "(" *(ctext / quoted-pair / comment) ")"
CR          =  <ASCII CR, carriage return>  ; (     15,      13.)
CRLF        =  CR LF
ctext       =  <any CHAR excluding "(",     ; => may be folded
                ")", "\" & CR, & including
                linear-white-space>
CTL         =  <any ASCII control           ; (  0- 37,  0.- 31.)
                character and DEL>          ; (    177,     127.)
date        =  1*2DIGIT month 2DIGIT        ; day month year
                                            ;  e.g. 20 Jun 82
dates       =   orig-date                   ; Original
              [ resent-date ]               ; Forwarded
date-time   =  [ day "," ] date time        ; dd mm yy
                                            ;  hh:mm:ss zzz
```

```
day          =  "Mon"  / "Tue" /  "Wed"  / "Thu"
             /  "Fri"  / "Sat" /  "Sun"
delimiters  =  specials / linear-white-space / comment
destination =  "To"            ":" 1#address  ; Primary
             /  "Resent-To"    ":" 1#address
             /  "cc"           ":" 1#address  ; Secondary
             /  "Resent-cc"    ":" 1#address
             /  "bcc"          ":"  #address  ; Blind carbon
             /  "Resent-bcc"   ":"  #address
DIGIT        =  <any ASCII decimal digit>     ; ( 60- 71, 48.- 57.)
domain       =  sub-domain *("." sub-domain)
domain-literal =  "[" *(dtext / quoted-pair) "]"
domain-ref   =  atom                          ; symbolic reference
dtext        =  <any CHAR excluding "[",      ; => may be folded
                 "]", "\" & CR, & including
                 linear-white-space>
extension-field =
               <Any field which is defined in a document
                published as a formal extension to this
                specification; none will have names beginning
                with the string "X-">
field        =  field-name ":" [ field-body ] CRLF
fields       =    dates                       ; Creation time,
                  source                      ;  author id & one
                1*destination                 ;  address required
                 *optional-field              ;  others optional
field-body   =  field-body-contents
                [CRLF LWSP-char field-body]
field-body-contents =
               <the ASCII characters making up the field-body, as
                defined in the following sections, and consisting
                of combinations of atom, quoted-string, and
                specials tokens, or else consisting of texts>
field-name   =  1*<any CHAR, excluding CTLs, SPACE, and ":">
group        =  phrase ":" [#mailbox] ";"
hour         =  2DIGIT ":" 2DIGIT [":" 2DIGIT]
                                              ; 00:00:00 - 23:59:59
HTAB         =  <ASCII HT, horizontal-tab>    ; (      11,        9.)
LF           =  <ASCII LF, linefeed>          ; (      12,       10.)
linear-white-space =  1*([CRLF] LWSP-char)    ; semantics = SPACE
                                              ; CRLF => folding
local-part   =  word *("." word)              ; uninterpreted
                                              ; case-preserved
```

```
LWSP-char    =  SPACE / HTAB                 ; semantics = SPACE
mailbox      =  addr-spec                    ; simple address
             /  phrase route-addr            ; name & addr-spec
message      =  fields *( CRLF *text )       ; Everything after
                                             ;  first null line
                                             ;  is message body
month        =  "Jan"  /  "Feb" /  "Mar"  /  "Apr"
             /  "May"  /  "Jun" /  "Jul"  /  "Aug"
             /  "Sep"  /  "Oct" /  "Nov"  /  "Dec"
msg-id       =  "<" addr-spec ">"            ; Unique message id
optional-field =
             /  "Message-ID"        ":"   msg-id
             /  "Resent-Message-ID" ":"   msg-id
             /  "In-Reply-To"       ":"  *(phrase / msg-id)
             /  "References"        ":"  *(phrase / msg-id)
             /  "Keywords"          ":"  #phrase
             /  "Subject"           ":"  *text
             /  "Comments"          ":"  *text
             /  "Encrypted"         ":" 1#2word
             /  extension-field              ; To be defined
             /  user-defined-field           ; May be pre-empted
orig-date    =  "Date"        ":"   date-time
originator   =   authentic                   ; authenticated addr
               [ "Reply-To"   ":" 1#address] )
phrase       =  1*word                       ; Sequence of words
qtext        =  <any CHAR excepting <">,     ; => may be folded
                 "\" & CR, and including
                linear-white-space>
quoted-pair =  "\" CHAR                      ; may quote any char
quoted-string = <"> *(qtext/quoted-pair) <">; Regular qtext or
                                             ;   quoted chars.
received     =  "Received"    ":"            ; one per relay
                   ["from" domain]           ; sending host
                   ["by"   domain]           ; receiving host
                   ["via"  atom]             ; physical path
                  *("with" atom)             ; link/mail protocol
                   ["id"   msg-id]           ; receiver msg id
                   ["for"  addr-spec]        ; initial form
                    ";"    date-time         ; time received
resent       =   resent-authentic
               [ "Resent-Reply-To"  ":" 1#address] )
resent-authentic =
             =   "Resent-From"      ":"   mailbox
             / ( "Resent-Sender"    ":"   mailbox
                 "Resent-From"      ":" 1#mailbox  )
resent-date =  "Resent-Date" ":"   date-time
return      =  "Return-path" ":" route-addr ; return address
route       =  1#("@" domain) ":"           ; path-relative
route-addr  =  "<" [route] addr-spec ">"
source      = [  trace ]                    ; net traversals
                 originator                 ; original mail
              [  resent ]                   ; forwarded
SPACE       =  <ASCII SP, space>            ; (     40,      32.)
specials    =  "(" / ")" / "<" / ">" / "@"  ; Must be in quoted-
            /  "," / ";" / ":" / "\" / <">  ;  string, to use
            /  "." / "[" / "]"              ;  within a word.
sub-domain  =  domain-ref / domain-literal
text        =  <any CHAR, including bare    ; => atoms, specials,
                CR & bare LF, but NOT       ;  comments and
                including CRLF>             ;  quoted-strings are
                                            ;  NOT recognized.
time        =  hour zone                    ; ANSI and Military
trace       =    return                     ; path to sender
               1*received                   ; receipt tags
user-defined-field =
              <Any field which has not been defined
               in this specification or published as an
               extension to this specification; names for
               such fields must be unique and may be
               pre-empted by published extensions>
word        =  atom / quoted-string

zone        =  "UT"  / "GMT"                ; Universal Time
                                            ; North American : UT
            /  "EST" / "EDT"                ;  Eastern:  - 5/ - 4
            /  "CST" / "CDT"                ;  Central:  - 6/ - 5
            /  "MST" / "MDT"                ;  Mountain: - 7/ - 6
            /  "PST" / "PDT"                ;  Pacific:  - 8/ - 7
            /  1ALPHA                       ; Military: Z = UT;
<">         =  <ASCII quote mark>           ; (     42,      34.)
```

Appendix B

PEM SYNTAX

B.1 pem_syntax.theory

```
Theory: pem_syntax

Parents:
    string
    HOL

Type constants:
    preeb 0
    posteb 0
    pemtypes 0
    proctype 0
    contentdescrip 0
    contentdomain 0
    algid 0
    IV 0
    dekinfo 0
    certificate 0
    id_asymmetric 0
    Key_info 0
    origid_asymm 0
    MIC_info 0

Term constants:
    is_preeb (Prefix)   :string -> bool
    REP_preeb (Prefix)   :preeb -> string
    BEGIN (Prefix)   :string -> preeb
    is_posteb (Prefix)   :string -> bool
    REP_posteb (Prefix)   :posteb -> string
    END (Prefix)   :string -> posteb
    REP_pemtypes (Prefix)   :pemtypes -> (one + one + one + one + one) ltree
    ABS_pemtypes (Prefix)   :(one + one + one + one + one) ltree -> pemtypes
    ENCRYPTED (Prefix)   :pemtypes
    MIC_ONLY (Prefix)   :pemtypes
    MIC_CLEAR (Prefix)   :pemtypes
    CRL (Prefix)   :pemtypes
    CRL_RETRIEVAL_REQUEST (Prefix)   :pemtypes
    is_proctype (Prefix)   :num # pemtypes -> bool
    REP_proctype (Prefix)   :proctype -> num # pemtypes
    Proc_Type (Prefix)   :num # pemtypes -> proctype
    REP_contentdescrip (Prefix)   :contentdescrip -> one ltree
    ABS_contentdescrip (Prefix)   :one ltree -> contentdescrip
    RFC822 (Prefix)   :contentdescrip
    REP_contentdomain (Prefix)   :contentdomain -> contentdescrip ltree
    ABS_contentdomain (Prefix)   :contentdescrip ltree -> contentdomain
    Content_Domain (Prefix)   :contentdescrip -> contentdomain
    REP_algid (Prefix)   :algid -> (one + one + one + one + one + one) ltree
    ABS_algid (Prefix)   :(one + one + one + one + one + one) ltree -> algid
    DES_CBC (Prefix)   :algid
    DES_EDE (Prefix)   :algid
    DES_ECB (Prefix)   :algid
    RSA (Prefix)   :algid
    RSA_MD2 (Prefix)   :algid
    RSA_MD5 (Prefix)   :algid
    REP_IV (Prefix)   :IV -> one ltree
    ABS_IV (Prefix)   :one ltree -> IV
    IV (Prefix)   :IV
    is_dekinfo (Prefix)   :algid # IV -> bool
    REP_dekinfo (Prefix)   :dekinfo -> algid # IV
    DEK_Info (Prefix)   :algid # IV -> dekinfo
    REP_certificate (Prefix)   :certificate -> string ltree
    ABS_certificate (Prefix)   :string ltree -> certificate
    Certificate (Prefix)   :string -> certificate
    REP_id_asymmetric (Prefix)   :id_asymmetric -> string ltree
    ABS_id_asymmetric (Prefix)   :string ltree -> id_asymmetric
    ID_Asymmetric (Prefix)   :string -> id_asymmetric
    is_Key_info (Prefix)   :algid # string -> bool
    REP_Key_info (Prefix)   :Key_info -> algid # string
    Key_Info (Prefix)   :algid # string -> Key_info
    REP_origid_asymm (Prefix)   :origid_asymm -> (one + one) ltree
    ABS_origid_asymm (Prefix)   :(one + one) ltree -> origid_asymm
    certificate (Prefix)   :origid_asymm
    id_asymmetric (Prefix)   :origid_asymm
    is_MIC_info (Prefix)   :algid # algid # string -> bool
    REP_MIC_info (Prefix)   :MIC_info -> algid # algid # string
    MIC_Info (Prefix)   :algid # algid # string -> MIC_info

Axioms:


Definitions:
    is_preeb |- !s. is_preeb s = s = 'PRIVACY-ENHANCED MESSAGE''
    preeb_TY_DEF |- ?rep. TYPE_DEFINITION is_preeb rep
    preeb_ISO_DEF
    |- (!a. BEGIN (REP_preeb a) = a) /\
       (!r. is_preeb r = REP_preeb (BEGIN r) = r)
    is_posteb |- !s. is_posteb s = s = 'PRIVACY-ENHANCED MESSAGE''
    posteb_TY_DEF |- ?rep. TYPE_DEFINITION is_posteb rep
    posteb_ISO_DEF
    |- (!a. END (REP_posteb a) = a) /\
       (!r. is_posteb r = REP_posteb (END r) = r)
    pemtypes_TY_DEF
    |- ?rep.
         TYPE_DEFINITION
          (TRP
           (\v tl.
             (v = INL one) /\ (LENGTH tl = 0) \/
             (v = INR (INL one)) /\ (LENGTH tl = 0) \/
             (v = INR (INR (INL one))) /\ (LENGTH tl = 0) \/
             (v = INR (INR (INR (INL one)))) /\ (LENGTH tl = 0) \/
             (v = INR (INR (INR (INR one)))) /\ (LENGTH tl = 0)))
          rep
   pemtypes_ISO_DEF
   |- (!a. ABS_pemtypes (REP_pemtypes a) = a) /\
      (!r.
        TRP
         (\v tl.
           (v = INL one) /\ (LENGTH tl = 0) \/
           (v = INR (INL one)) /\ (LENGTH tl = 0) \/
           (v = INR (INR (INL one))) /\ (LENGTH tl = 0) \/
           (v = INR (INR (INR (INL one)))) /\ (LENGTH tl = 0) \/
           (v = INR (INR (INR (INR one)))) /\ (LENGTH tl = 0))
        r =
        REP_pemtypes (ABS_pemtypes r) =
        r)
ENCRYPTED_DEF |- ENCRYPTED = ABS_pemtypes (Node (INL one) [])
MIC_ONLY_DEF |- MIC_ONLY = ABS_pemtypes (Node (INR (INL one)) [])
MIC_CLEAR_DEF |- MIC_CLEAR = ABS_pemtypes (Node (INR (INR (INL one))) [])
CRL_DEF |- CRL = ABS_pemtypes (Node (INR (INR (INR (INL one)))) [])
CRL_RETRIEVAL_REQUEST_DEF
|- CRL_RETRIEVAL_REQUEST =
   ABS_pemtypes (Node (INR (INR (INR (INR one)))) [])
is_proctype |- !proctype. is_proctype proctype = FST proctype = 4
proctype_TY_DEF |- ?rep. TYPE_DEFINITION is_proctype rep
proctype_ISO_DEF
|- (!a. Proc_Type (REP_proctype a) = a) /\
   (!r. is_proctype r = REP_proctype (Proc_Type r) = r)
contentdescrip_TY_DEF
|- ?rep. TYPE_DEFINITION (TRP (\v tl. (v = one) /\ (LENGTH tl = 0))) rep
contentdescrip_ISO_DEF
|- (!a. ABS_contentdescrip (REP_contentdescrip a) = a) /\
   (!r.
      TRP (\v tl. (v = one) /\ (LENGTH tl = 0)) r =
      REP_contentdescrip (ABS_contentdescrip r) =
      r)
RFC822_DEF |- RFC822 = ABS_contentdescrip (Node one [])
contentdomain_TY_DEF
|- ?rep. TYPE_DEFINITION (TRP (\v tl. (?c. v = c) /\ (LENGTH tl = 0))) rep
contentdomain_ISO_DEF
|- (!a. ABS_contentdomain (REP_contentdomain a) = a) /\
   (!r.
      TRP (\v tl. (?c. v = c) /\ (LENGTH tl = 0)) r =
      REP_contentdomain (ABS_contentdomain r) =
      r)
Content_Domain_DEF |- !c. Content_Domain c = ABS_contentdomain (Node c [])
algid_TY_DEF
|- ?rep.
      TYPE_DEFINITION
        (TRP
          (\v tl.
            (v = INL one) /\ (LENGTH tl = 0) \/
            (v = INR (INL one)) /\ (LENGTH tl = 0) \/
            (v = INR (INR (INL one))) /\ (LENGTH tl = 0) \/
            (v = INR (INR (INR (INL one)))) /\ (LENGTH tl = 0) \/
            (v = INR (INR (INR (INR (INL one))))) /\ (LENGTH tl = 0) \/
            (v = INR (INR (INR (INR (INR one))))) /\ (LENGTH tl = 0)))
        rep
algid_ISO_DEF
|- (!a. ABS_algid (REP_algid a) = a) /\
   (!r.
      TRP
        (\v tl.
          (v = INL one) /\ (LENGTH tl = 0) \/
          (v = INR (INL one)) /\ (LENGTH tl = 0) \/
          (v = INR (INR (INL one))) /\ (LENGTH tl = 0) \/
          (v = INR (INR (INR (INL one)))) /\ (LENGTH tl = 0) \/
          (v = INR (INR (INR (INR (INL one))))) /\ (LENGTH tl = 0) \/
          (v = INR (INR (INR (INR (INR one))))) /\ (LENGTH tl = 0))
        r =
      REP_algid (ABS_algid r) =
      r)
DES_CBC_DEF |- DES_CBC = ABS_algid (Node (INL one) [])
DES_EDE_DEF |- DES_EDE = ABS_algid (Node (INR (INL one)) [])
DES_ECB_DEF |- DES_ECB = ABS_algid (Node (INR (INR (INL one))) [])
RSA_DEF |- RSA = ABS_algid (Node (INR (INR (INR (INL one)))) [])
RSA_MD2_DEF
|- RSA_MD2 = ABS_algid (Node (INR (INR (INR (INR (INL one))))) [])
```

```
RSA_MD5_DEF
|- RSA_MD5 = ABS_algid (Node (INR (INR (INR (INR (INR one))))) [])
IV_TY_DEF
|- ?rep. TYPE_DEFINITION (TRP (\v tl. (v = one) /\ (LENGTH tl = 0))) rep
IV_ISO_DEF
|- (!a. ABS_IV (REP_IV a) = a) /\
   (!r.
     TRP (\v tl. (v = one) /\ (LENGTH tl = 0)) r = REP_IV (ABS_IV r) = r)
IV_DEF |- IV = ABS_IV (Node one [])
is_dekinfo |- !a. is_dekinfo a = FST a = DES_CBC
dekinfo_TY_DEF |- ?rep. TYPE_DEFINITION is_dekinfo rep
dekinfo_ISO_DEF
|- (!a. DEK_Info (REP_dekinfo a) = a) /\
   (!r. is_dekinfo r = REP_dekinfo (DEK_Info r) = r)
certificate_TY_DEF
|- ?rep. TYPE_DEFINITION (TRP (\v tl. (?s. v = s) /\ (LENGTH tl = 0))) rep
certificate_ISO_DEF
|- (!a. ABS_certificate (REP_certificate a) = a) /\
   (!r.
     TRP (\v tl. (?s. v = s) /\ (LENGTH tl = 0)) r =
     REP_certificate (ABS_certificate r) =
     r)
Certificate_DEF |- !s. Certificate s = ABS_certificate (Node s [])
id_asymmetric_TY_DEF
|- ?rep. TYPE_DEFINITION (TRP (\v tl. (?s. v = s) /\ (LENGTH tl = 0))) rep
id_asymmetric_ISO_DEF
|- (!a. ABS_id_asymmetric (REP_id_asymmetric a) = a) /\
   (!r.
     TRP (\v tl. (?s. v = s) /\ (LENGTH tl = 0)) r =
     REP_id_asymmetric (ABS_id_asymmetric r) =
     r)
ID_Asymmetric_DEF |- !s. ID_Asymmetric s = ABS_id_asymmetric (Node s [])
is_Key_info |- !x. is_Key_info x = FST x = RSA
Key_info_TY_DEF |- ?rep. TYPE_DEFINITION is_Key_info rep
Key_info_ISO_DEF
|- (!a. Key_Info (REP_Key_info a) = a) /\
   (!r. is_Key_info r = REP_Key_info (Key_Info r) = r)
origid_asymm_TY_DEF
|- ?rep.
     TYPE_DEFINITION
       (TRP
         (\v tl.
           (v = INL one) /\ (LENGTH tl = 0) \/
           (v = INR one) /\ (LENGTH tl = 0)))
       rep
origid_asymm_ISO_DEF
|- (!a. ABS_origid_asymm (REP_origid_asymm a) = a) /\
   (!r.
     TRP
       (\v tl.
         (v = INL one) /\ (LENGTH tl = 0) \/
         (v = INR one) /\ (LENGTH tl = 0))
       r =
     REP_origid_asymm (ABS_origid_asymm r) =
     r)
certificate_DEF |- certificate = ABS_origid_asymm (Node (INL one) [])
id_asymmetric_DEF |- id_asymmetric = ABS_origid_asymm (Node (INR one) [])
is_MIC_info
|- !x.
     is_MIC_info x =
     ((FST x = RSA_MD2) \/ (FST x = RSA_MD5)) /\
     ((FST (SND x) = DES_EDE) \/
      (FST (SND x) = DES_ECB) \/
      (FST (SND x) = RSA))
```

```
    MIC_info_TY_DEF |- ?rep. TYPE_DEFINITION is_MIC_info rep
    MIC_info_ISO_DEF
    |- (!a. MIC_Info (REP_MIC_info a) = a) /\
       (!r. is_MIC_info r = REP_MIC_info (MIC_Info r) = r)

Theorems:
    REP_preeb_INVERTS |- !a. BEGIN (REP_preeb a) = a
    REP_preeb_ONE_ONE |- !a a'. (REP_preeb a = REP_preeb a') = a = a'
    REP_preeb_ONTO |- !r. is_preeb r = (?a. r = REP_preeb a)
    ABS_preeb_INVERTS |- !r. is_preeb r = REP_preeb (BEGIN r) = r
    ABS_preeb_ONE_ONE
    |- !r r'. is_preeb r ==> is_preeb r' ==> ((BEGIN r = BEGIN r') = r = r')
    ABS_preeb_ONTO |- !a. ?r. (a = BEGIN r) /\ is_preeb r
    REP_posteb_INVERTS |- !a. END (REP_posteb a) = a
    REP_posteb_ONE_ONE |- !a a'. (REP_posteb a = REP_posteb a') = a = a'
    REP_posteb_ONTO |- !r. is_posteb r = (?a. r = REP_posteb a)
    ABS_posteb_INVERTS |- !r. is_posteb r = REP_posteb (END r) = r
    ABS_posteb_ONE_ONE
    |- !r r'. is_posteb r ==> is_posteb r' ==> ((END r = END r') = r = r')
    ABS_posteb_ONTO |- !a. ?r. (a = END r) /\ is_posteb r
    pemtypes
    |- !e0 e1 e2 e3 e4.
         ?!fn.
           (fn ENCRYPTED = e0) /\
           (fn MIC_ONLY = e1) /\
           (fn MIC_CLEAR = e2) /\
           (fn CRL = e3) /\
           (fn CRL_RETRIEVAL_REQUEST = e4)
    pemtypes_INDUCT
    |- !P.
         P ENCRYPTED /\
         P MIC_ONLY /\
         P MIC_CLEAR /\
         P CRL /\
         P CRL_RETRIEVAL_REQUEST ==>
         (!p. P p)
    pemtypes_DISTINCT
    |- ~(ENCRYPTED = MIC_ONLY) /\
       ~(ENCRYPTED = MIC_CLEAR) /\
       ~(ENCRYPTED = CRL) /\
       ~(ENCRYPTED = CRL_RETRIEVAL_REQUEST) /\
       ~(MIC_ONLY = MIC_CLEAR) /\
       ~(MIC_ONLY = CRL) /\
       ~(MIC_ONLY = CRL_RETRIEVAL_REQUEST) /\
       ~(MIC_CLEAR = CRL) /\
       ~(MIC_CLEAR = CRL_RETRIEVAL_REQUEST) /\
       ~(CRL = CRL_RETRIEVAL_REQUEST)
    pemtypes_CASES
    |- !p.
         (p = ENCRYPTED) \/
         (p = MIC_ONLY) \/
         (p = MIC_CLEAR) \/
         (p = CRL) \/
         (p = CRL_RETRIEVAL_REQUEST)
    REP_proctype_INVERTS |- !a. Proc_Type (REP_proctype a) = a
    REP_proctype_ONE_ONE |- !a a'. (REP_proctype a = REP_proctype a') = a = a'
    REP_proctype_ONTO |- !r. is_proctype r = (?a. r = REP_proctype a)
    ABS_proctype_INVERTS |- !r. is_proctype r = REP_proctype (Proc_Type r) = r
    ABS_proctype_ONE_ONE
    |- !r r'.
         is_proctype r ==>
         is_proctype r' ==>
         ((Proc_Type r = Proc_Type r') = r = r')
    ABS_proctype_ONTO |- !a. ?r. (a = Proc_Type r) /\ is_proctype r
```

```
contentdescrip |- !e. ?!fn. fn RFC822 = e
contentdescrip_INDUCT |- !P. P RFC822 ==> (!c. P c)
contentdescrip_CASES |- !c. c = RFC822
contentdomain |- !f. ?!fn. !c. fn (Content_Domain c) = f c
contentdomain_INDUCT |- !P. (!c. P (Content_Domain c)) ==> (!c. P c)
contentdomain_CASES |- !c. ?c'. c = Content_Domain c'
algid
|- !e0 e1 e2 e3 e4 e5.
     ?!fn.
       (fn DES_CBC = e0) /\
       (fn DES_EDE = e1) /\
       (fn DES_ECB = e2) /\
       (fn RSA = e3) /\
       (fn RSA_MD2 = e4) /\
       (fn RSA_MD5 = e5)
algid_INDUCT
|- !P.
     P DES_CBC /\
     P DES_EDE /\
     P DES_ECB /\
     P RSA /\
     P RSA_MD2 /\
     P RSA_MD5 ==>
     (!a. P a)
algid_DISTINCT
|- ~(DES_CBC = DES_EDE) /\
   ~(DES_CBC = DES_ECB) /\
   ~(DES_CBC = RSA) /\
   ~(DES_CBC = RSA_MD2) /\
   ~(DES_CBC = RSA_MD5) /\
   ~(DES_EDE = DES_ECB) /\
   ~(DES_EDE = RSA) /\
   ~(DES_EDE = RSA_MD2) /\
   ~(DES_EDE = RSA_MD5) /\
   ~(DES_ECB = RSA) /\
   ~(DES_ECB = RSA_MD2) /\
   ~(DES_ECB = RSA_MD5) /\
   ~(RSA = RSA_MD2) /\
   ~(RSA = RSA_MD5) /\
   ~(RSA_MD2 = RSA_MD5)
algid_CASES
|- !a.
     (a = DES_CBC) \/
     (a = DES_EDE) \/
     (a = DES_ECB) \/
     (a = RSA) \/
     (a = RSA_MD2) \/
     (a = RSA_MD5)
IV |- !e. ?!fn. fn IV = e
REP_dekinfo_INVERTS |- !a. DEK_Info (REP_dekinfo a) = a
REP_dekinfo_ONE_ONE |- !a a'. (REP_dekinfo a = REP_dekinfo a') = a = a'
REP_dekinfo_ONTO |- !r. is_dekinfo r = (?a. r = REP_dekinfo a)
ABS_dekinfo_INVERTS |- !r. is_dekinfo r = REP_dekinfo (DEK_Info r) = r
ABS_dekinfo_ONE_ONE
|- !r r'.
     is_dekinfo r ==>
     is_dekinfo r' ==>
     ((DEK_Info r = DEK_Info r') = r = r')
ABS_dekinfo_ONTO |- !a. ?r. (a = DEK_Info r) /\ is_dekinfo r
certificate |- !f. ?!fn. !s. fn (Certificate s) = f s
id_assymmetric |- !f. ?!fn. !s. fn (ID_Asymmetric s) = f s
REP_Key_info_INVERTS |- !a. Key_Info (REP_Key_info a) = a
REP_Key_info_ONE_ONE |- !a a'. (REP_Key_info a = REP_Key_info a') = a = a'
REP_Key_info_ONTO |- !r. is_Key_info r = (?a. r = REP_Key_info a)
```

```
ABS_Key_info_INVERTS |- !r. is_Key_info r = REP_Key_info (Key_Info r) = r
ABS_Key_info_ONE_ONE
|- !r r'.
     is_Key_info r ==>
     is_Key_info r' ==>
     ((Key_Info r = Key_Info r') = r = r')
ABS_Key_info_ONTO |- !a. ?r. (a = Key_Info r) /\ is_Key_info r
origid_asymm
|- !e0 e1. ?!fn. (fn certificate = e0) /\ (fn id_asymmetric = e1)
REP_MIC_info_INVERTS |- !a. MIC_Info (REP_MIC_info a) = a
REP_MIC_info_ONE_ONE |- !a a'. (REP_MIC_info a = REP_MIC_info a') = a = a'
REP_MIC_info_ONTO |- !r. is_MIC_info r = (?a. r = REP_MIC_info a)
ABS_MIC_info_INVERTS |- !r. is_MIC_info r = REP_MIC_info (MIC_Info r) = r
ABS_MIC_info_ONE_ONE
|- !r r'.
     is_MIC_info r ==>
     is_MIC_info r' ==>
     ((MIC_Info r = MIC_Info r') = r = r')
ABS_MIC_info_ONTO |- !a. ?r. (a = MIC_Info r) /\ is_MIC_info r
```

B.2 pem_syntax.sml

```
(*======================================================*)
(* File:        pem_syntax.sml                          *)
(* Description: PEM message syntax                      *)
(* Date:        July 2, 1996                            *)
(* Author:      Shiu-Kai Chin, with small modification  *)
(*              by Dan Zhou                             *)
(*======================================================*)


load_library{lib = hol88_lib, theory = "-"};
open Psyntax Compat;

(* Definition of PEM messages.          *)
new_theory "pem_syntax";
new_parent "string";
use "/amd/humbolt/sw/hol90.7/library/string/src/ascii_conv.sml";
use "/amd/humbolt/sw/hol90.7/library/string/src/string_conv.sml";
use "/amd/humbolt/sw/hol90.7/library/string/src/string_rules.sml";
open String_rules;


add_definitions_to_sml "string";
add_theorems_to_sml "string";

add_theory_to_sml "pair";
add_definitions_to_sml "pair";
add_theory_to_sml "list";
add_definitions_to_sml "list";

(*=     =       =       =       =       =       =       =       =*)
(* Define pre-encapsulation boundary *)

val is_preeb =  new_definition ("is_preeb",(--`is_preeb(s:string)
        = (s = "PRIVACY-ENHANCED MESSAGE")`--));

val exists_preeb = TAC_PROOF(
        ([],--`?s.is_preeb s`--),
        EXISTS_TAC (--`"PRIVACY-ENHANCED MESSAGE"`--) THEN
        REWRITE_TAC [is_preeb]);
```

```
val preeb_TY_DEF = new_type_definition("preeb",
        (--`is_preeb`--),exists_preeb);

val preeb_ISO_DEF = define_new_type_bijections
        "preeb_ISO_DEF" "BEGIN" "REP_preeb" preeb_TY_DEF;

val REP_preeb_INVERTS =
        save_thm("REP_preeb_INVERTS",CONJUNCT1 preeb_ISO_DEF);

val REP_preeb_ONE_ONE =
        save_thm("REP_preeb_ONE_ONE",prove_rep_fn_one_one
        preeb_ISO_DEF);

val REP_preeb_ONTO =
        save_thm("REP_preeb_ONTO", prove_rep_fn_onto preeb_ISO_DEF);

val ABS_preeb_INVERTS =
        save_thm("ABS_preeb_INVERTS", CONJUNCT2 preeb_ISO_DEF);

val ABS_preeb_ONE_ONE =
        save_thm("ABS_preeb_ONE_ONE",prove_abs_fn_one_one preeb_ISO_DEF);

val ABS_preeb_ONTO =
        save_thm("ABS_preeb_ONTO", prove_abs_fn_onto preeb_ISO_DEF);


(*=       =       =       =       =       =       =       =       =*)
(* Define post-encapsulation boundary *)

val is_posteb = new_definition(
        "is_posteb",(--`is_posteb(s:string) =
        (s = "PRIVACY-ENHANCED MESSAGE")`--));

val exists_posteb = TAC_PROOF(
        ([],--`?s.is_posteb s`--),
        EXISTS_TAC (--`"PRIVACY-ENHANCED MESSAGE"`--) THEN
        REWRITE_TAC [is_posteb]);

val posteb_TY_DEF = new_type_definition("posteb",
        (--`is_posteb`--),exists_posteb);

val posteb_ISO_DEF = define_new_type_bijections
        "posteb_ISO_DEF" "END" "REP_posteb" posteb_TY_DEF;

val REP_posteb_INVERTS =
        save_thm("REP_posteb_INVERTS",CONJUNCT1 posteb_ISO_DEF);

val REP_posteb_ONE_ONE =
        save_thm("REP_posteb_ONE_ONE",prove_rep_fn_one_one
        posteb_ISO_DEF);

val REP_posteb_ONTO =
        save_thm("REP_posteb_ONTO", prove_rep_fn_onto posteb_ISO_DEF);

val ABS_posteb_INVERTS =
        save_thm("ABS_posteb_INVERTS", CONJUNCT2 posteb_ISO_DEF);

val ABS_posteb_ONE_ONE =
        save_thm("ABS_posteb_ONE_ONE",prove_abs_fn_one_one
        posteb_ISO_DEF);

val ABS_posteb_ONTO =
        save_thm("ABS_posteb_ONTO", prove_abs_fn_onto posteb_ISO_DEF);
```

```
(*:     :       :       :       :       :       :       :       :*)
(* we will just take pemtext as a string, so there is no need   *)
(* to have a separate type for it                               *)


(*:     :       :       :       :       :       :       :       :*)
(* Definitions of header structures *)

(* Define the message types. *)
val pemtypes = define_type
        {name="pemtypes",
        fixities=[Prefix,Prefix,Prefix,Prefix,Prefix],
        type_spec = `pemtypes = ENCRYPTED | MIC_ONLY | MIC_CLEAR
                | CRL | CRL_RETRIEVAL_REQUEST`};

val pemtypes_INDUCT  =
        save_thm("pemtypes_INDUCT",prove_induction_thm pemtypes);

val pemtypes_DISTINCT =
        save_thm("pemtypes_DISTINCT",prove_constructors_distinct
        pemtypes);

val pemtypes_CASES =
        save_thm("pemtypes_CASES", prove_cases_thm pemtypes_INDUCT);


(*:     :       :       :       :       :       :       :       :*)
(* The Proc_type field has two subfields.  The first is a number *)
(* identifying the version of PEM.  The second identifies the    *)
(* type of security used.                                        *)

(* Define the subset of pairs *)
val is_proctype = new_definition
        ("is_proctype",
        --`is_proctype(proctype:(num#pemtypes))
        = (FST proctype = 4)`--);

add_theorems_to_sml "pair";

(* Show at least one element exists in the type *)
val exists_proctype = TAC_PROOF(
        ([],(--`?x:(num#pemtypes).is_proctype x`--)),
        EXISTS_TAC (--`(4,ENCRYPTED)`--)
        THEN REWRITE_TAC [is_proctype,FST]);

val proctype_TY_DEF = new_type_definition("proctype",
        (--`is_proctype`--),exists_proctype);

val proctype_ISO_DEF = define_new_type_bijections
        "proctype_ISO_DEF" "Proc_Type" "REP_proctype" proctype_TY_DEF;

val REP_proctype_INVERTS =
        save_thm("REP_proctype_INVERTS",CONJUNCT1 proctype_ISO_DEF);

val REP_proctype_ONE_ONE =
        save_thm("REP_proctype_ONE_ONE",prove_rep_fn_one_one
        proctype_ISO_DEF);

val REP_proctype_ONTO =
        save_thm("REP_proctype_ONTO", prove_rep_fn_onto
        proctype_ISO_DEF);
```

```
val ABS_proctype_INVERTS :
        save_thm(''ABS_proctype_INVERTS'', CONJUNCT2 proctype_ISO_DEF);

val ABS_proctype_ONE_ONE :
        save_thm(''ABS_proctype_ONE_ONE'',prove_abs_fn_one_one
        proctype_ISO_DEF);

val ABS_proctype_ONTO :
        save_thm(''ABS_proctype_ONTO'', prove_abs_fn_onto
        proctype_ISO_DEF);


(*:     :       :       :       :       :       :       :       :*)
(* Definition of contentdescrip *)

val contentdescrip : define_type
        {name : ''contentdescrip'',
        fixities : [Prefix],
        type_spec : `contentdescrip : RFC822`};

val contentdescrip_INDUCT  :
        save_thm(''contentdescrip_INDUCT',prove_induction_thm
        contentdescrip);

val contentdescrip_CASES :
        save_thm(''contentdescrip_CASES'', prove_cases_thm
        contentdescrip_INDUCT);


(*:     :       :       :       :       :       :       :       :*)
(* Definition of contentdomain *)

val contentdomain : define_type
        {name : ''contentdomain'',
        fixities : [Prefix],
        type_spec : `contentdomain : Content_Domain of
                contentdescrip`};

val contentdomain_INDUCT  :
        save_thm(''contentdomain_INDUCT'',prove_induction_thm
                contentdomain);

val contentdomain_CASES :
        save_thm(''contentdomain_CASES'', prove_cases_thm
                contentdomain_INDUCT);


(*:     :       :       :       :       :       :       :       :*)
(* Definitions of algid *)

val algid : define_type
        {name : ''algid'',
        fixities : [Prefix, Prefix, Prefix, Prefix, Prefix, Prefix],
        type_spec : `algid : DES_CBC | DES_EDE | DES_ECB | RSA
                | RSA_MD2 | RSA_MD5`};

val algid_INDUCT  :
        save_thm(''algid_INDUCT',prove_induction_thm algid);

val algid_DISTINCT :
        save_thm(''algid_DISTINCT',prove_constructors_distinct algid);

val algid_CASES :
        save_thm(''algid_CASES'', prove_cases_thm algid_INDUCT);
```

```
(*:     :       :       :       :       :       :       :       :*)
(* Fake dekparameters -- just 16 hex characters for an initialization
   vector *)

val IV : define_type
        {name : ''IV'',
        fixities : [Prefix],
        type_spec : `IV : IV`};


(*:     :       :       :       :       :       :       :       :*)
(* Definition of dekinfo *)

val is_dekinfo : new_definition(''is_dekinfo'',
        (--`is_dekinfo(a:(algid#IV)) : (FST a : DES_CBC)`--));

val exists_dekinfo : TAC_PROOF(
        ([],(--`?a.is_dekinfo(a)`--)),
        EXISTS_TAC (--`(DES_CBC,IV)`--) THEN
        REWRITE_TAC [is_dekinfo,FST]);

val dekinfo_TY_DEF : new_type_definition(''dekinfo'',
        (--`is_dekinfo`--),exists_dekinfo);

val dekinfo_ISO_DEF :define_new_type_bijections
        ''dekinfo_ISO_DEF'' ''DEK_Info'' ''REP_dekinfo'' dekinfo_TY_DEF;

val REP_dekinfo_INVERTS :
        save_thm(''REP_dekinfo_INVERTS'',CONJUNCT1 dekinfo_ISO_DEF);

val REP_dekinfo_ONE_ONE :
        save_thm(''REP_dekinfo_ONE_ONE'',prove_rep_fn_one_one
                dekinfo_ISO_DEF);

val REP_dekinfo_ONTO :
        save_thm(''REP_dekinfo_ONTO'', prove_rep_fn_onto
                dekinfo_ISO_DEF);

val ABS_dekinfo_INVERTS :
        save_thm(''ABS_dekinfo_INVERTS'', CONJUNCT2 dekinfo_ISO_DEF);

val ABS_dekinfo_ONE_ONE :
        save_thm(''ABS_dekinfo_ONE_ONE'',prove_abs_fn_one_one
                dekinfo_ISO_DEF);

val ABS_dekinfo_ONTO :
        save_thm(''ABS_dekinfo_ONTO'', prove_abs_fn_onto
                dekinfo_ISO_DEF);


(*:     :       :       :       :       :       :       :       :*)
(* Definition of certificate *)
(* cert - fake it for now as a string *)
val certificate : define_type
        {name : ''certificate'',
        fixities : [Prefix],
        type_spec : `certificate : Certificate of string`};
(* since we don't really use certificate right now, will jsut   *)
(* leave it here  - 4/18/96 *)


(*:     :       :       :       :       :       :       :       :*)
```

```
(* Definition of ID Asymmetric *)
(* id_asymmetric - fake it for now as a string *)
val id_asymmetric : define_type
        {name : "id_assymmetric",
        fixities : [Prefix],
        type_spec : `id_asymmetric : ID_Asymmetric of string`};


(*:     :       :       :       :       :       :       :       :*)
(* Key-Info *)
(* this is the per-message encrypted by each recipient's public key *)

val is_Key_info :new_definition
        ("is_Key_info", (--`is_Key_info(x:algid#string) :
                                                        (* asymsgKey *)
        (FST x) : RSA`--));

val exists_Key_info : TAC_PROOF(
        ([],(--`?x:algid#string. is_Key_info(x)`--)),
                        (* asymsgKey *)
        EXISTS_TAC (--`(RSA,"abced")`--) THEN
                        (* asymsgKey *)
        REWRITE_TAC [is_Key_info, FST, SND]);

val Key_info_TY_DEF : new_type_definition("Key_info",
        (--`is_Key_info`--),exists_Key_info);

val Key_info_ISO_DEF :  define_new_type_bijections
        "Key_info_ISO_DEF" "Key_Info" "REP_Key_info" Key_info_TY_DEF;

val REP_Key_info_INVERTS :
        save_thm("REP_Key_info_INVERTS",CONJUNCT1 Key_info_ISO_DEF);

val REP_Key_info_ONE_ONE :
        save_thm("REP_Key_info_ONE_ONE",prove_rep_fn_one_one
        Key_info_ISO_DEF);

val REP_Key_info_ONTO :
        save_thm("REP_Key_info_ONTO", prove_rep_fn_onto
        Key_info_ISO_DEF);

val ABS_Key_info_INVERTS :
        save_thm("ABS_Key_info_INVERTS", CONJUNCT2 Key_info_ISO_DEF);

val ABS_Key_info_ONE_ONE :
        save_thm("ABS_Key_info_ONE_ONE",prove_abs_fn_one_one
        Key_info_ISO_DEF);

val ABS_Key_info_ONTO :
        save_thm("ABS_Key_info_ONTO", prove_abs_fn_onto
        Key_info_ISO_DEF);


(*:     :       :       :       :       :       :       :       :*)
(* Definitions for origIds -- just asymmetric for now *)
(* asymmid - it's either a certificate or id_asymmetric *)
val origid_asymm : define_type
        {name : "origid_asymm",
        fixities : [Prefix,Prefix],
        type_spec : `origid_asymm : certificate | id_asymmetric`};


(*:     :       :       :       :       :       :       :       :*)
(* Definitions for MIC_info *)
```

```
val is_MIC_info =new_definition
        ("is_MIC_info", (--`is_MIC_info(x:algid#algid#string) =
                                                  (* asymsignmic *)
        (((FST x) = RSA_MD2) \/ ((FST x) = RSA_MD5)) /\
        (((FST(SND x)) = DES_EDE) \/ ((FST(SND x)) = DES_ECB)
        \/ ((FST(SND x)) = RSA))`--));

val exists_MIC_info = TAC_PROOF(
        ([],(--`?x:algid#algid#string. is_MIC_info(x)`--)),
                                (* asymsignmic *)
        EXISTS_TAC (--`(RSA_MD2,DES_EDE,"abced")`--) THEN
                                        (* asymsignmic *)
        REWRITE_TAC [is_MIC_info, FST, SND]);

val MIC_info_TY_DEF = new_type_definition("MIC_info",
        (--`is_MIC_info`--),exists_MIC_info);

val MIC_info_ISO_DEF =  define_new_type_bijections
        "MIC_info_ISO_DEF" "MIC_Info" "REP_MIC_info" MIC_info_TY_DEF;

val REP_MIC_info_INVERTS =
        save_thm("REP_MIC_info_INVERTS",CONJUNCT1 MIC_info_ISO_DEF);

val REP_MIC_info_ONE_ONE =
        save_thm("REP_MIC_info_ONE_ONE",prove_rep_fn_one_one
        MIC_info_ISO_DEF);

val REP_MIC_info_ONTO =
        save_thm("REP_MIC_info_ONTO", prove_rep_fn_onto
        MIC_info_ISO_DEF);

val ABS_MIC_info_INVERTS =
        save_thm("ABS_MIC_info_INVERTS", CONJUNCT2 MIC_info_ISO_DEF);

val ABS_MIC_info_ONE_ONE =
        save_thm("ABS_MIC_info_ONE_ONE",prove_abs_fn_one_one
        MIC_info_ISO_DEF);

val ABS_MIC_info_ONTO =
        save_thm("ABS_MIC_info_ONTO", prove_abs_fn_onto
        MIC_info_ISO_DEF);


(*=     =      =      =      =      =      =      =      =*)
(* issuer's certificate *)


(*=     =      =      =      =      =      =      =      =*)
(* recipient information *)
(* a recipient information is: (id_asymmetric#Key_info),        *)
(* all recipient information is: (id_asymmetric#Key_info) list  *)


close_theory();
export_theory();
```

Appendix C

PEM_DEFINITIONS

C.1 pem_definitions.theory

```
Theory: pem_definitions

Parents:
    pem_syntax

Type constants:


Term constants:
    msgreceiver (Prefix)   :string
    recipientkey (Prefix)   :string
    sDES_EDE (Prefix)   :string -> string -> string -> bool
    sDES_ECB (Prefix)   :string -> string -> string -> bool
    sRSA (Prefix)   :string -> string -> string -> bool
    fRSA (Prefix)   :string -> string -> string
    fRSA_MD2 (Prefix)   :string -> string
    fRSA_MD5 (Prefix)   :string -> string
    fDES_CBC (Prefix)   :string -> string -> IV -> string
    get_Key_from_ID (Prefix)   :id_asymmetric -> string
    get_DEK_algid (Prefix)   :dekinfo -> algid
    get_DEK_IV (Prefix)   :dekinfo -> IV
    msg_Encrypt_select (Prefix)   :dekinfo -> string -> string -> IV -> string
    get_Recipient (Prefix)
    :string list -> (id_asymmetric # string) list -> id_asymmetric # string
    get_Recipient_key (Prefix)   :id_asymmetric # string -> string
    get_Recipient_asyID (Prefix)   :id_asymmetric # string -> id_asymmetric
    get_MIC_algid (Prefix)   :MIC_info -> algid
    get_MIC_sigalgid (Prefix)   :MIC_info -> algid
    get_MIC_mic (Prefix)   :MIC_info -> string
    MIC_hash_select (Prefix)   :MIC_info -> string -> string
    MIC_sign_select (Prefix)   :MIC_info -> string -> string -> string -> bool
    get_KEY_algid (Prefix)   :Key_info -> algid
    get_KEY_asymsgKey (Prefix)   :Key_info -> string
    DEK_encrypt_select (Prefix)   :Key_info -> string -> string -> string
    is_PrivateS (Prefix)
    :(string -> string -> IV -> string) -> string -> string -> IV -> string ->
     bool
    is_PrivateP (Prefix)
    :(string -> string -> string) -> string -> string -> string -> bool
    is_Authentic (Prefix)
    :(string -> string -> string -> bool) -> string -> string -> string -> bool
    is_Authentic2 (Prefix)
    :(string -> string -> string -> bool) -> (string -> string) -> string ->
     string -> string -> bool
    is_Intact (Prefix)
    :(string -> string -> string -> bool) -> (string -> string) -> string ->
     string -> string -> bool
    is_non_deniable (Prefix)
    :(string -> string -> string -> bool) -> string -> string -> string -> bool

Axioms:


Definitions:
    get_DEK_algid |- !x. get_DEK_algid x = FST (REP_dekinfo x)
    get_DEK_IV |- !x. get_DEK_IV x = SND (REP_dekinfo x)
    msg_Encrypt_select
    |- !x.
         msg_Encrypt_select x =
         ((get_DEK_algid x = DES_CBC) => fDES_CBC | fDES_CBC)
    get_Recipient_key
    |- !recipient. get_Recipient_key recipient = SND recipient
    get_Recipient_asyID
    |- !recipient. get_Recipient_asyID recipient = FST recipient
    get_MIC_algid |- !x. get_MIC_algid x = FST (REP_MIC_info x)
    get_MIC_sigalgid |- !x. get_MIC_sigalgid x = FST (SND (REP_MIC_info x))
    get_MIC_mic |- !x. get_MIC_mic x = SND (SND (REP_MIC_info x))
    MIC_hash_select
    |- !x.
         MIC_hash_select x =
         ((get_MIC_algid x = RSA_MD2) => fRSA_MD2 | fRSA_MD5)
    MIC_sign_select
    |- !x.
         MIC_sign_select x =
         ((get_MIC_sigalgid x = DES_EDE)
          => sDES_EDE
          | ((get_MIC_sigalgid x = DES_ECB) => sDES_ECB | sRSA))
    get_KEY_algid |- !x. get_KEY_algid x = FST (REP_Key_info x)
    get_KEY_asymsgKey |- !x. get_KEY_asymsgKey x = SND (REP_Key_info x)
    DEK_encrypt_select
    |- !x. DEK_encrypt_select x = ((get_KEY_algid x = RSA) => fRSA | fRSA)
    is_PrivateS
    |- !decryptS message rxmsg decryptIV key.
         is_PrivateS decryptS message rxmsg decryptIV key =
         decryptS rxmsg key decryptIV =
         message
    is_PrivateP
    |- !decryptP message rxmsg dkey.
         is_PrivateP decryptP message rxmsg dkey =
         decryptP rxmsg dkey =
         message
    is_Authentic
    |- !verify message signature ekey.
         is_Authentic verify message signature ekey =
         verify message signature ekey
    is_Authentic2
    |- !verify hash message mic ekey.
         is_Authentic2 verify hash message mic ekey =
         verify (hash message) mic ekey
    is_Intact
    |- !verify hash message mic ekey.
         is_Intact verify hash message mic ekey =
         verify (hash message) mic ekey
    is_non_deniable
    |- !verify message signature ekey.
         is_non_deniable verify message signature ekey =
         verify message signature ekey

Theorems:
    is_Private_DEK
    |- !decryptP encryptP message txmsg rxmsg ekey dKEY0 dkey.
         (rxmsg = txmsg) ==>
```

```
      (txmsg = encryptP message ekey) ==>
      (!msg. decryptP (encryptP msg ekey) dKEY0 = msg) ==>
      (!msg d2.
        (decryptP (encryptP msg ekey) d2 = msg) ==> (d2 = dKEY0)) ==>
      ((dkey = dKEY0) = is_PrivateP decryptP message rxmsg dkey)
  is_Private_msg
  |- !decryptS encryptS message txmsg rxmsg decryptIV KEY0 key.
      (rxmsg = txmsg) ==>
      (txmsg = encryptS message KEY0 decryptIV) ==>
      (!msg key.
        (decryptS (encryptS msg key decryptIV) key decryptIV = msg) /\
        (!msg key1.
          (decryptS msg key1 decryptIV = decryptS msg key decryptIV) =
          key =
          key1)) ==>
      ((key = KEY0) = is_PrivateS decryptS message rxmsg decryptIV key)
  is_Authentic_MD
  |- !verify sign message txmsg rxmsg ekey dKEY0 dkey.
      (rxmsg = txmsg) ==>
      (txmsg = sign MD dkey) ==>
      (!msg. verify msg (sign msg dkey) ekey = dkey = dKEY0) ==>
      ((dkey = dKEY0) = is_Authentic verify MD rxmsg ekey)
  is_Authentic_msg
  |- !verify sign hash message txmic rxmic ekey dKEY0 dkey.
      (rxmic = txmic) ==>
      (txmic = sign (hash message) dkey) ==>
      (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey = dkey2 = dKEY0) ==>
      ((dkey = dKEY0) = is_Authentic2 verify hash message rxmic ekey)
  is_Intact_msg
  |- !verify sign hash txmessage rxmessage txmic rxmic ekey dkey.
      (txmic = sign (hash txmessage) dkey) ==>
      (rxmic = txmic) ==>
      (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
      (!s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ==>
      ((rxmessage = txmessage) = is_Intact verify hash rxmessage rxmic ekey)
  is_non_deniable_msg
  |- !verify sign hash message MESSAGE0 txmic rxmic ekey dKEY0 dkey.
      (rxmic = txmic) ==>
      (txmic = sign (hash MESSAGE0) dkey) ==>
      (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
      (!m1 m2 dkey2.
        verify m1 (sign m2 dkey2) ekey = (m1 = m2) /\ (dkey2 = dKEY0)) ==>
      ((dkey = dKEY0) /\ (message = MESSAGE0) =
       is_non_deniable verify (hash message) rxmic ekey)
  not_Authentic
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
      (txmic = sign (hash MESSAGE0) dKEY0) ==>
      (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
      (!m1 m2 dkey1 dkey2.
        (sign m1 dkey1 = sign m2 dkey2) ==>
        (m1 = m2) /\ (dkey1 = dkey2)) ==>
      ~(rxmic = txmic) ==>
      ~(is_Authentic2 verify hash MESSAGE0 rxmic ekey)
  not_Intact
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
      (txmic = sign (hash MESSAGE0) dKEY0) ==>
      (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
      (!m1 m2 dkey1 dkey2.
        (sign m1 dkey1 = sign m2 dkey2) ==>
        (m1 = m2) /\ (dkey1 = dkey2)) ==>
      ~(rxmic = txmic) ==>
      ~(is_Intact verify hash MESSAGE0 rxmic ekey)
  is_deniable
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
```

```
        (txmic = sign (hash MESSAGE0) dKEY0) ==>
        (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
        (!m1 m2 dkey1 dkey2.
          (sign m1 dkey1 = sign m2 dkey2) ==>
          (m1 = m2) /\ (dkey1 = dkey2)) ==>
        ~(rxmic = txmic) ==>
        ~(is_non_deniable verify (hash MESSAGE0) rxmic ekey)
    get_DEK_algid_CASES |- !x. get_DEK_algid x = DES_CBC
    get_MIC_hashid_CASES
    |- !x. (get_MIC_algid x = RSA_MD2) \/ (get_MIC_algid x = RSA_MD5)
    get_MIC_signid_CASES
    |- !x.
        (get_MIC_sigalgid x = DES_EDE) \/
        (get_MIC_sigalgid x = DES_ECB) \/
        (get_MIC_sigalgid x = RSA)
    get_Key_algid_CASES |- !x. get_KEY_algid x = RSA
```

C.2 pem_definitions.sml

```
(*==================================================*)
(* File:         pem_definitions.sml                *)
(* Description:  general functions for PEM          *)
(* Date:         Sept. 13, 1996                     *)
(* Author:       Shiu-Kai Chin,  Dan Zhou           *)
(*==================================================*)


(* msgsender:    the actual sender of the message              *)
(* Originator:   the stated sender in the message              *)
(* msgreceiver:  the actual receiver of the message,           *)
(*               the one that performs PEM services            *)
(* Recipient:    the intended recipient of the message         *)
(* verify:       takes msg, signature, and key                 *)
(* message:      plain text                                    *)
(* msg:          ciper text                                    *)
(* mic:          message integrity code, or digital signature  *)
(* encryptS:     plaintext -> ekey -> IV -> ciphertext         *)

new_theory "pem_definitions";

load_library{lib = hol88_lib, theory = "-"};
open Psyntax Compat;

new_parent "pem_syntax";

add_theory_to_sml "pem_syntax";

val msgreceiver = new_constant ("msgreceiver",
        ==`:string`==);

(* the private key of recipient                                *)
val recipientkey = new_constant ("recipientkey",
        ==`:string`==);

val sDES_EDE = new_constant
        ("sDES_EDE", ==`:string->string->string->bool`==);
val sDES_ECB = new_constant
        ("sDES_ECB", ==`:string->string->string->bool`==);
val sRSA = new_constant
        ("sRSA", ==`:string->string->string->bool`==);
```

```
val fRSA = new_constant ("fRSA", ==`:string->string->string`==);

val fRSA_MD2 = new_constant ("fRSA_MD2", ==`:string->string`==);
val fRSA_MD5 = new_constant ("fRSA_MD5", ==`:string->string`==);

val fDES_CBC = new_constant ("fDES_CBC",
        ==`:string->string->IV->string`==);

val get_Key_from_ID = new_constant
        ("get_Key_from_ID", ==`:id_asymmetric->string`==);


(*=     =       =       =       =       =       =       =*)
(* these are the algorith ID and IV for encrypting/decrypting   *)
(* message                                                      *)
val get_DEK_algid = new_definition ("get_DEK_algid",
        (--`get_DEK_algid (x:dekinfo) =
        FST(REP_dekinfo x)`--));

val get_DEK_IV = new_definition ("get_DEK_IV",
        (--`get_DEK_IV (x:dekinfo) =
        SND(REP_dekinfo x)`--));

val msg_Encrypt_select = new_definition ("msg_Encrypt_select",
        (--`msg_Encrypt_select (x:dekinfo) =
        ((get_DEK_algid x = DES_CBC) => fDES_CBC | fDES_CBC)`--));

(*=     =       =       =       =       =       =       =       =*)
val get_Recipient = new_constant (
        "get_Recipient",
        ==`:(string list) ->
        ((id_asymmetric#string) list) -> (id_asymmetric#string)`==);


val get_Recipient_key = new_definition ("get_Recipient_key",
        --`get_Recipient_key (recipient:id_asymmetric#string)
        = SND recipient`--);

val get_Recipient_asyID = new_definition ("get_Recipient_asyID",
        --`get_Recipient_asyID (recipient:id_asymmetric#string)
        = FST recipient`--);

(*=     =       =       =       =       =       =       =       =*)
val get_MIC_algid = new_definition ("get_MIC_algid",
        (--`get_MIC_algid (x:MIC_info) =
        FST(REP_MIC_info x)`--));
val get_MIC_sigalgid = new_definition ("get_MIC_sigalgid",
        (--`get_MIC_sigalgid (x:MIC_info)
        = FST(SND(REP_MIC_info x))`--));
val get_MIC_mic = new_definition ("get_MIC_mic",
        (--`get_MIC_mic (x:MIC_info)
        = SND(SND(REP_MIC_info x))`--));

val MIC_hash_select = new_definition ("MIC_hash_select",
        (--`(MIC_hash_select:MIC_info->(string->string))
        (x:MIC_info) =
        ((get_MIC_algid x = RSA_MD2) => fRSA_MD2 | fRSA_MD5)`--));

val MIC_sign_select = new_definition ("MIC_sign_select",
        (--`(MIC_sign_select:
                MIC_info->(string->string->string->bool))
        (x:MIC_info) =
        ((get_MIC_sigalgid x = DES_EDE) => sDES_EDE |
        ((get_MIC_sigalgid x = DES_ECB) => sDES_ECB | sRSA))`--));
```

```
(*:     :     :     :     :     :     :     :       :*)
(* encrypted DEK information                          *)
val get_KEY_algid : new_definition (``get_KEY_algid'',
        (--`get_KEY_algid (x:Key_info) :
        FST(REP_Key_info x)`--));

val get_KEY_asymsgKey : new_definition (``get_KEY_asymsgKey'',
        (--`get_KEY_asymsgKey (x:Key_info) :
        SND(REP_Key_info x)`--));

val DEK_encrypt_select : new_definition (``DEK_encrypt_select'',
        (--`(DEK_encrypt_select:Key_info->(string->string->string))
        (x:Key_info) :
        ((get_KEY_algid x : RSA) :> fRSA | fRSA)`--));


(*:     :     :     :     :     :     :     :       :*)
(* key convention:                                    *)
(*      in public key cryptography: ekey: public key  *)
(*                                  dkey: private key *)
(*      in secret key cryptography: key, ekey, dkey: same thing *)

(* term convention:                                   *)

(* MIC: a fixed-length quantity generated cryptographically  *)
(* and associated with a message to reassure the recipient that *)
(* the message is genuine                             *)

(* digital signature: same for MIC, in public key case *)

(* signature: a quantity asocited with a message which only *)
(* someone with knowledge of your private key could have  *)
(* generated, but which can be verified through knowledge of *)
(* your public key                                    *)


(* if you can retrieve the original message by decryption, then *)
(* you are the intended recipient                     *)
val is_PrivateS : new_definition (``is_PrivateS'',
       (--`is_PrivateS
       (decryptS: string -> string -> IV ->string)
       (message: string) (* plain text *)
       (rxmsg:   string) (* cipher text *)
       (decryptIV:  IV)
       (key: string) :
              (decryptS rxmsg key decryptIV : message)`--));

val is_PrivateP : new_definition (``is_PrivateP'',
       (--`is_PrivateP
       (decryptP: string -> string ->string)
       (message: string) (* plain text *)
       (rxmsg:   string) (* cipher text *)
       (dkey: string) :
              (decryptP rxmsg dkey : message)`--));


(* is_Authentic: if I can check the signature, then only   *)
(* the person who knows the private key could have signed  *)
(* the text                                            *)
val is_Authentic : new_definition (``is_Authentic'',
       (--`is_Authentic
        (verify: string -> string -> string -> bool)
        (message: string)   (* plain text *)
        (signature: string) (* signature of message *)
        (ekey: string) =
                verify message signature ekey`--));


(* is_Authentic2: if I can check the digital signature of a      *)
(* messge, then only the person who knows the private key could *)
(* have signed the text                                          *)
val is_Authentic2 = new_definition ("is_Authentic2",
        (--`is_Authentic2
        (verify: string -> string -> string -> bool)
        (hash: string -> string)
        (message: string) (* original plain text *)
        (mic:string) (* received signature of the MD*)
        (ekey: string) =
                verify (hash message) mic ekey`--));


(* if you can verifying the signature of a message digest,       *)
(* then you can be sure if the message is intact                 *)
val is_Intact = new_definition ("is_Intact",
        (--`is_Intact
        (verify:string -> string -> string -> bool)
        (hash: string -> string)
        (message:string)
        (mic:string)
        (ekey:string)  =
                verify (hash message) mic ekey`--));


(* a private key uniquely identifies with a principal           *)
(* so if the message is signed with an dkey, then only the      *)
(* owner of dkey would have signed it                           *)

val is_non_deniable = new_definition ("is_non_deniable",
        (--`is_non_deniable
        (verify: string -> string -> string -> bool)
        (message: string) (* original plain text *)
        (signature:string) (* received signature *)
        (ekey: string) =
                verify message signature ekey`--));

close_theory();
export_theory();


(*=     =       =       =       =       =       =       =       =*)
(* prove the property is_private                                *)

(* the per message key is secure                                *)
(*val is_Private_DEK =
|- !decryptP encryptP message txmsg rxmsg ekey dKEY0 dkey.
        (rxmsg = txmsg) ==>
        (txmsg = encryptP message ekey) ==>
        (!msg. decryptP (encryptP msg ekey) dKEY0 = msg) ==>
        (!msg d2. (decryptP (encryptP msg ekey) d2 = msg)
                ==> (d2 = dKEY0)) ==>
        ((dkey = dKEY0) = is_PrivateP decryptP message rxmsg dkey)
*)

val is_Private_DEK = prove_thm ("is_Private_DEK",
        (--`!(decryptP:string->string->string)
        (encryptP:string->string->string)
        (message: string) (* plaintext  *)
        (txmsg:string)    (* ciphertext *)
        (rxmsg:string)    (* ciphertext *)
        (ekey: string)
        (dKEY0: string)
        (dkey: string).
        (rxmsg = txmsg) ==>
        (txmsg = encryptP message ekey) ==>
        (!msg. (decryptP (encryptP msg ekey) dKEY0) = msg) ==>
        (!msg d2. ((decryptP (encryptP msg ekey) d2) = msg)
              ==> (d2 = dKEY0)) ==>
        ((dkey = dKEY0) = is_PrivateP decryptP message rxmsg dkey)`--),
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_PrivateP]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`message:string`--] th)) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`message:string`--, --`dkey:string`--] th)) THEN
        EQ_TAC THENL
        [DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        ASM_REWRITE_TAC [],
        PURE_ONCE_ASM_REWRITE_TAC []]);


(*val is_Private_msg =
|- !decryptS encryptS message txmsg rxmsg decryptIV KEY0 key.
        (rxmsg = txmsg) ==>
        (txmsg = encryptS message KEY0 decryptIV) ==>
        (!msg key.
           (decryptS (encryptS msg key decryptIV) key decryptIV = msg)
           /\ (!msg key1. (decryptS msg key1 decryptIV
                 = decryptS msg key decryptIV) = key = key1)) ==>
        ((key = KEY0)
                = is_PrivateS decryptS message rxmsg decryptIV key)
*)

val is_Private_msg =
         prove_thm (``is_Private_msg'',
        (--`!(decryptS: string -> string -> IV -> string)
        (encryptS: string -> string -> IV -> string)
        (message: string) (* plaintext  *)
        (txmsg: string)    (* ciphertext *)
        (rxmsg: string)    (* ciphertext *)
        (decryptIV: IV)
        (KEY0: string)
        (key: string).
        (rxmsg = txmsg) ==>
        (txmsg = encryptS message KEY0 decryptIV) ==>
        (!msg key. (decryptS
                (encryptS msg key decryptIV) key decryptIV = msg) /\
        !msg key1. ((decryptS msg key1 decryptIV
                = decryptS msg key decryptIV) = key = key1)) ==>
        ((key = KEY0) =
                is_PrivateS decryptS message rxmsg decryptIV key)`--),
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_PrivateS]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => MP_TAC
          (SPECL [--`message: string`--, --`KEY0: string`--] th)) THEN
        DISCH_THEN (fn th => ASSUME_TAC (CONJUNCT1 th) THEN
          MP_TAC (SPECL
            [--`(encryptS: string -> string -> IV -> string)
```

```
              message KEY0 decryptIV`--,
                --`key:string`--] (CONJUNCT2 th))) THEN
        DISCH_THEN (fn th => ASSUME_TAC th) THEN
        EQ_TAC THENL
        [DISCH_THEN (fn th => ASM_REWRITE_TAC [th]),
        UNDISCH_TAC (--`(decryptS: string -> string -> IV -> string)
                (encryptS message KEY0 decryptIV)
                KEY0 decryptIV = message`--) THEN
        DISCH_THEN (fn th1 => (DISCH_THEN (fn th2 => ASSUME_TAC (GSYM(
          (SUBST [((GSYM th2), --`x:string`--)]
            (--`(decryptS: string -> string -> IV -> string)
              (encryptS (message: string) KEY0 decryptIV)
                KEY0 decryptIV= x`--) th1))))))
        THEN
        RES_TAC THEN
        ASM_REWRITE_TAC []]);


(*:     :       :       :       :       :       :       :       :*)
(* prove the property is_authentic *)
(*val is_Authentic_MD :
|- !verify sign message txmsg rxmsg ekey dKEY0 dkey.
        (rxmsg = txmsg) ==>
        (txmsg = sign MD dkey) ==>
        (!msg. verify msg (sign msg dkey) ekey = dkey = dKEY0) ==>
        ((dkey = dKEY0) = is_Authentic verify MD rxmsg ekey)
*)

val is_Authentic_MD = prove_thm ("is_Authentic_MD",
        (--`!(verify:string->string->string->bool)
        (sign:string->string->string)
        (message: string) (* plaintext  *)
        (txmsg:string)    (* ciphertext *)
        (rxmsg:string)    (* ciphertext *)
        (ekey: string)
        (dKEY0: string)
        (dkey: string).
        (rxmsg = txmsg) ==>
        (txmsg = sign MD dkey) ==>
        (!msg. verify msg (sign msg dkey) ekey = dkey = dKEY0) ==>
        ((dkey = dKEY0) =
        is_Authentic verify MD rxmsg ekey)`--),
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_Authentic]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`MD:string`--] th)) THEN
        ASM_REWRITE_TAC []);

 (* assure the recipient that the sender did send the message    *)
 (* val is_Authentic_msg =
 |- !verify sign hash message txmic rxmic ekey dKEY0 dkey.
        (rxmic = txmic) ==>
        (txmic = sign (hash message) dkey) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey = dkey2 = dKEY0) ==>
        ((dkey = dKEY0) = is_Authentic2 verify hash message rxmic ekey)
 *)

 val is_Authentic_msg = prove_thm ("is_Authentic_msg",
        (--`!(verify:string->string->string->bool)
        (sign :string->string->string)
        (hash: string->string)
        (message :string)  (* plaintext  *)
        (txmic :string)     (* digital signature *)
        (rxmic :string)     (* digital signature *)
        (ekey: string)
        (dKEY0: string)
        (dkey: string).
        (rxmic = txmic) ==>
        (txmic = sign (hash message) dkey) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = (dkey2 = dKEY0)) ==>
        ((dkey = dKEY0) =
        is_Authentic2 verify hash message rxmic ekey)`--),
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_Authentic2]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`(hash:string->string) (message:string)`--,
             --`(hash:string->string) (message:string)`--,
             --`dkey:string`--] th)) THEN
        ASM_REWRITE_TAC []);


(*=       =       =       =       =       =       =       =        =*)

(* is_Intact applied to a message,                                 *)
(*val is_Intact_msg =
|- !verify sign hash txmessage rxmessage txmic rxmic ekey dkey.
        (txmic = sign (hash txmessage) dkey) ==>
        (rxmic = txmic) ==>
        (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ==>
        ((rxmessage = txmessage)
          = is_Intact verify hash rxmessage rxmic ekey)
*)

val is_Intact_msg = prove_thm ("is_Intact_msg",
        --`!(verify:string -> string -> string ->bool)
        (sign:string -> string -> string)
        (hash:  string-> string)
        (txmessage: string ) (rxmessage: string)
        (txmic: string) (rxmic: string)
        (ekey: string) (dkey: string).
        (txmic = sign (hash txmessage) dkey) ==>
        (rxmic = txmic) ==>
        (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!s1 s2. verify s1 (sign s2 dkey) ekey = (s1 = s2)) ==>
        ((rxmessage = txmessage) =
          is_Intact verify hash rxmessage rxmic ekey)`--,
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_Intact]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC (SPECL
          [(--`rxmessage:string`--), (--`txmessage:string`--)] th)) THEN
        DISCH_THEN (fn th => ASSUME_TAC (SPECL
          [(--`(hash:string->string) (txmessage:string)`--),
          (--`(hash:string->string) (txmessage:string)`--)] th)
          THEN MP_TAC th) THEN
        DISCH_THEN (fn th => ASSUME_TAC (SPECL
          [(--`(hash:string->string) (rxmessage:string)`--),
          (--`(hash:string->string) (txmessage:string)`--)] th)) THEN
        EQ_TAC THENL
        [DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        ASM_REWRITE_TAC [],
        ASM_REWRITE_TAC []]);
```

```
(*=     =       =       =       =       =       =       =       =*)
(* prove non-repudiation                                          *)
(*val is_non_deniable_msg =
|- !verify sign hash message MESSAGE0 txmic rxmic ekey dKEY0 dkey.
        (rxmic = txmic) ==>
        (txmic = sign (hash MESSAGE0) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = (m1 = m2) /\ (dkey2 = dKEY0)) ==>
        ((dkey = dKEY0) /\ (message = MESSAGE0) =
                is_non_deniable verify (hash message) rxmic ekey)
*)

val is_non_deniable_msg = prove_thm ("is_non_deniable_msg",
        (--`!(verify:string->string->string->bool)
        (sign :string->string->string)
        (hash: string->string)
        (message :string)  (* plaintext, retrieved by recipient *)
        (MESSAGE0: string) (* plaintext, used by originator *)
        (txmic :string)
        (rxmic :string)
        (ekey: string)  (* public key of claimed originator *)
        (dKEY0: string) (* private key of claimed originator *)
        (dkey: string). (* private key of real originator *)
        (rxmic = txmic) ==>
        (txmic = sign (hash MESSAGE0) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = ((m1 = m2) /\ (dkey2 = dKEY0))) ==>
        (((dkey = dKEY0) /\ (message = MESSAGE0)) =
        is_non_deniable verify (hash message) rxmic ekey)`--),
        REPEAT GEN_TAC THEN
        DISCH_THEN (fn th => REWRITE_TAC [th, is_non_deniable]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`message:string`--,--`MESSAGE0:string`--] th)) THEN
        DISCH_THEN (fn th => ASSUME_TAC
          (SPECL [--`(hash:string->string) (message:string)`--,
            --`(hash:string->string) (MESSAGE0:string)`--,
            --`dkey:string`--] th)) THEN
        ASM_REWRITE_TAC [] THEN
        ACCEPT_TAC (SPECL [--`(dkey:string) = (dKEY0:string)`--,
          --`(message:string) = (MESSAGE0:string)`--] CONJ_SYM));


 (*=     =       =       =       =       =       =       =       =*)

 val th = TAC_PROOF (
        ([], --`!A B. (~A==>~B) = ( B==>A)`--),
        REPEAT GEN_TAC THEN EQ_TAC THENL
        [DISCH_THEN (fn th => MP_TAC(IMP_ELIM th)) THEN
        SUBST1_TAC (SPECL [--`~~A`--, --`~B`--] DISJ_SYM) THEN
        DISCH_THEN (fn th => MP_TAC (DISJ_IMP th)) THEN
        REWRITE_TAC [NOT_CLAUSES],
        DISCH_THEN (fn th => MP_TAC(IMP_ELIM th)) THEN
        SUBST1_TAC (SPECL [--`~B`--, --`A:bool`--] DISJ_SYM) THEN
        DISCH_THEN (fn th => MP_TAC (DISJ_IMP th)) THEN
        REWRITE_TAC [NOT_CLAUSES]]);

 (*=     =       =       =       =       =       =       =       =       =*)

 (* This says that if I send you a message and the MIC is somehow
    changed on the way, then you cannot be sure of the source of the
```

```
   message *)

(* val not_Authentic :
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
       (txmic = sign (hash MESSAGE0) dKEY0) ==>
       (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
       (!m1 m2 dkey1 dkey2.
         (sign m1 dkey1 = sign m2 dkey2) ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
       ~(rxmic = txmic) ==>
       ~(is_Authentic2 verify hash MESSAGE0 rxmic ekey) : thm
*)

val not_Authentic = prove_thm ("not_Authentic",
        (--`!(verify:string->string->string->bool)
        (sign :string->string->string)
        (hash: string->string)
        (MESSAGE0: string) (* plaintext, used by originator *)
        (txmic :string)
        (rxmic :string)
        (ekey: string)   (* public key of claimed originator *)
        (dKEY0: string). (* private key of claimed originator *)
        (txmic = sign (hash MESSAGE0) dKEY0) ==>
        (!m1 m2. verify m1 m2 ekey = (m2 = sign m1 dKEY0)) ==>
        (!m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
                ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
        ~(rxmic = txmic) ==>
        ~(is_Authentic2 verify hash MESSAGE0 rxmic ekey)`--),
        REPEAT GEN_TAC THEN
        REWRITE_TAC [is_Authentic2, th] THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]));


(*=     =       =       =       =       =       =       =       =        =*)
(* This says that if I send you a message and the MIC is somehow
   changed on the way, then you cannot be sure of the integrity of
   both MIC and message, since either one could have been changed *)

(* val not_Intact :
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
       (txmic = sign (hash MESSAGE0) dKEY0) ==>
       (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
       (!m1 m2 dkey1 dkey2.
         (sign m1 dkey1 = sign m2 dkey2) ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
       ~(rxmic = txmic) ==>
       ~(is_Intact verify hash MESSAGE0 rxmic ekey) : thm
*)

val not_Intact = prove_thm ("not_Intact",
        (--`!(verify:string->string->string->bool)
        (sign :string->string->string)
        (hash: string->string)
        (MESSAGE0: string) (* plaintext, used by originator *)
        (txmic :string)
        (rxmic :string)
        (ekey: string)   (* public key of claimed originator *)
        (dKEY0: string). (* private key of claimed originator *)
        (txmic = sign (hash MESSAGE0) dKEY0) ==>
        (!m1 m2. verify m1 m2 ekey = (m2 = sign m1 dKEY0)) ==>
        (!m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
                ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
        ~(rxmic = txmic) ==>
        ~(is_Intact verify hash MESSAGE0 rxmic ekey)`--),
        REPEAT GEN_TAC THEN
        REWRITE_TAC [is_Intact, th] THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]));


(*=     =        =        =        =        =        =        =        =        =*)
(* This says that if I send you a message and the MIC is somehow
   changed on the way, then I can deny having sent the message. *)
(* The reason we assume the received message is correct and the
   claimed identity of originator is real, is, otherwise, one cannot
   say that someone didn't send the mail, instead of this one didn't
   send this mail message *)

(* val is_deniable =
  |- !verify sign hash MESSAGE0 txmic rxmic ekey dKEY0.
       (txmic = sign (hash MESSAGE0) dKEY0) ==>
       (!m1 m2. verify m1 m2 ekey = m2 = sign m1 dKEY0) ==>
       (!m1 m2 dkey1 dkey2.
         (sign m1 dkey1 = sign m2 dkey2) ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
       ~(rxmic = txmic) ==>
       ~(is_non_deniable verify (hash MESSAGE0) rxmic ekey) : thm
*)

val is_deniable = prove_thm ("is_deniable",
        (--`!(verify:string->string->string->bool)
        (sign :string->string->string)
        (hash: string->string)
        (MESSAGE0: string) (* plaintext, used by originator *)
        (txmic :string)
        (rxmic :string)
        (ekey: string)   (* public key of claimed originator *)
        (dKEY0: string). (* private key of claimed originator *)
        (txmic = sign (hash MESSAGE0) dKEY0) ==>
        (!m1 m2. verify m1 m2 ekey = (m2 = sign m1 dKEY0)) ==>
        (!m1 m2 dkey1 dkey2. (sign m1 dkey1 = sign m2 dkey2)
                ==> (m1 = m2) /\ (dkey1 = dkey2)) ==>
        ~(rxmic = txmic) ==>
        ~(is_non_deniable verify (hash MESSAGE0) rxmic ekey)`--),
        REPEAT GEN_TAC THEN
        REWRITE_TAC [is_non_deniable, th] THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]) THEN
        DISCH_THEN (fn th => REWRITE_TAC [th]));


(*=     =        =        =        =        =        =        =        =*)
val th1 = SPECL [--`x:dekinfo`--]
        (CONJUNCT1 dekinfo_ISO_DEF);
val th2 = REWRITE_RULE [th1] (SPECL [--`REP_dekinfo (x:dekinfo)`--]
        (CONJUNCT2 dekinfo_ISO_DEF));
val th3 = REWRITE_RULE [is_dekinfo] th2;

(*val get_DEK_algid_CASES = |- !x. get_DEK_algid x = DES_CBC     *)

val get_DEK_algid_CASES = prove_thm ("get_DEK_algid_CASES",
        --`!x. (get_DEK_algid x = DES_CBC)`--,
        GEN_TAC THEN
        REWRITE_TAC [get_DEK_algid, th3]);


(*=     =        =        =        =*)
val th1 = SPECL [--`x:MIC_info`--]
        (CONJUNCT1 MIC_info_ISO_DEF);
val th2 = SPECL [--`REP_MIC_info (x:MIC_info)`--]
        (CONJUNCT2 MIC_info_ISO_DEF);
```

```
val th3 = REWRITE_RULE [th1] th2;
val th4 = REWRITE_RULE [is_MIC_info] th3;
val th5 = CONJUNCT1 th4;
val th6 = CONJUNCT2 th4;

(*
get_MIC_hashid_CASES:
|- !x. (get_MIC_algid x = RSA_MD2) \/ (get_MIC_algid x = RSA_MD5)
*)
val get_MIC_hashid_CASES = prove_thm (''get_MIC_hashid_CASES'',
          --'!x. (get_MIC_algid x = RSA_MD2 ) \/
                  (get_MIC_algid x = RSA_MD5)'--,

          GEN_TAC THEN
          REWRITE_TAC [get_MIC_algid, th5]);


(*get_MIC_signid_CASES:
|- !x.
        (get_MIC_sigalgid x = DES_EDE) \/
        (get_MIC_sigalgid x = DES_ECB) \/
        (get_MIC_sigalgid x = RSA) : thm
*)
val get_MIC_signid_CASES = prove_thm (''get_MIC_signid_CASES'',
          --'!x. (get_MIC_sigalgid x = DES_EDE) \/
                  (get_MIC_sigalgid x = DES_ECB) \/
                  (get_MIC_sigalgid x = RSA)'--,
          GEN_TAC THEN
          REWRITE_TAC [get_MIC_sigalgid, th6]);


(*=     =       =       =       =       =       =       =       =*)
val th1 = SPECL [--'x:Key_info'--]
          (CONJUNCT1 Key_info_ISO_DEF);
val th2 = REWRITE_RULE [th1] (SPECL [--'REP_Key_info (x:Key_info)'--]
          (CONJUNCT2 Key_info_ISO_DEF));
val th3 = REWRITE_RULE [is_Key_info] th2;


(*val get_Key_algid_CASES = |- !x. get_KEY_algid x = RSA        *)

val get_Key_algid_CASES = prove_thm (''get_Key_algid_CASES'',
          --'!x. (get_KEY_algid x = RSA)'--,
          GEN_TAC THEN
          REWRITE_TAC [get_KEY_algid, th3]);


(*=     =       =       =       =       =       =       =       =*)

export_theory();
```

Appendix D

PEM_CLEAR

D.1 pem_clear.theory

```
Theory: pem_clear

Parents:
    pem_definitions

Type constants:


Term constants:
    MIC_CLEAR_example (Prefix)
    :string -> string -> string -> string ->
     preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb
    get_MIC_CLEAR_MIC_Info (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> MIC_info
    get_OriginatorAsymID_info (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> id_asymmetric
    get_MIC_CLEAR_Proc_Type (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> proctype
    get_MIC_CLEAR_text (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> string
    get_msg_HashID (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> algid
    get_msg_SignID (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> algid
    get_msg_MIC (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> string
    MIC_CLEAR_is_Intact (Prefix)
    :preeb # proctype # contentdomain # id_asymmetric # certificate list #
     MIC_info # string # posteb -> bool

Axioms:


Definitions:
    MIC_CLEAR_example
    |- !s1 s2 s3 s4.
         MIC_CLEAR_example s1 s2 s3 s4 =
         (BEGIN 'PRIVACY_ENHANCED MAIL'',
          Proc_Type (4,MIC_CLEAR),
          Content_Domain RFC822,
          ID_Asymmetric s1,
          [Certificate s2],
          MIC_Info (RSA_MD5,RSA,s3),
          s4,
          END 'PRIVACY_ENHANCED MAIL'')
    get_MIC_CLEAR_MIC_Info
    |- !x. get_MIC_CLEAR_MIC_Info x = FST (SND (SND (SND (SND (SND x)))))
    get_OriginatorAsymID_info
    |- !x. get_OriginatorAsymID_info x = FST (SND (SND (SND x)))
    get_MIC_CLEAR_Proc_Type |- !x. get_MIC_CLEAR_Proc_Type x = FST (SND x)
    get_MIC_CLEAR_text
    |- !x. get_MIC_CLEAR_text x = FST (SND (SND (SND (SND (SND (SND x))))))
    get_msg_HashID
    |- !x. get_msg_HashID x = get_MIC_algid (get_MIC_CLEAR_MIC_Info x)
    get_msg_SignID
    |- !x. get_msg_SignID x = get_MIC_sigalgid (get_MIC_CLEAR_MIC_Info x)
    get_msg_MIC |- !x. get_msg_MIC x = get_MIC_mic (get_MIC_CLEAR_MIC_Info x)
    MIC_CLEAR_is_Intact
    |- !mic_clear_msg.
         MIC_CLEAR_is_Intact mic_clear_msg =
         (let micInfo = get_MIC_CLEAR_MIC_Info mic_clear_msg
          in
          let ekey = get_Key_from_ID (get_OriginatorAsymID_info mic_clear_msg)
          in
          is_Intact (MIC_sign_select micInfo) (MIC_hash_select micInfo)
            (get_MIC_CLEAR_text mic_clear_msg)
            (get_MIC_mic micInfo)
            ekey)

Theorems:
    integrity_lemma1
    |- !verify sign hash txmessage rxmessage dkey ekey.
         (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
         (!m1 m2. verify m1 (sign m2 dkey) ekey = m1 = m2) ==>
         is_Intact verify hash rxmessage (sign (hash txmessage) dkey) ekey ==>
         (txmessage = rxmessage)
    integrity_lemma2
    |- !verify sign hash txmessage rxmessage dkey ekey.
         (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
         (!m1 m2. verify m1 (sign m2 dkey) ekey = m1 = m2) ==>
         (txmessage = rxmessage) ==>
         is_Intact verify hash rxmessage (sign (hash txmessage) dkey) ekey
    integrity_lemma3
    |- !verify sign hash txmessage rxmessage dkey ekey.
         (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
         (!m1 m2. verify m1 (sign m2 dkey) ekey = m1 = m2) ==>
         ((txmessage = rxmessage) =
          is_Intact verify hash rxmessage (sign (hash txmessage) dkey) ekey)
    Intact
    |- !verify sign hash txmessage rxmessage dkey ekey smd.
         (smd = sign (hash txmessage) dkey) ==>
         (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
         (!m1 m2. verify m1 (sign m2 dkey) ekey = m1 = m2) ==>
         ((txmessage = rxmessage) = is_Intact verify hash rxmessage smd ekey)
    MIC_CLEAR_is_Intact_Correct
    |- !mic_clear_msg sign txmessage dkey.
         let micInfo = get_MIC_CLEAR_MIC_Info mic_clear_msg
         in
         let ekey = get_Key_from_ID (get_OriginatorAsymID_info mic_clear_msg)
         in
         let hash = MIC_hash_select micInfo
         and
         verify = MIC_sign_select micInfo
         and
```

```
        rxmessage : get_MIC_CLEAR_text mic_clear_msg
        in
        (get_MIC_mic micInfo = sign (hash txmessage) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2. verify m1 (sign m2 dkey) ekey = m1 = m2) ==>
        ((txmessage = rxmessage) = MIC_CLEAR_is_Intact mic_clear_msg)
```

D.2 pem_clear.sml

```
(*=====================================================*)
(* File:        pem_clear.sml                          *)
(* Description: selector and security function for     *)
(*              MIC-CLEAR message                      *)
(* Date:        Aug. 20, 1996                          *)
(* Author:      Shiu-Kai Chin,  with some modification *)
(*              by Dan Zhou                            *)
(*=====================================================*)

(*      sign: use private key, ''dkey''                *)
(*    verify: use public key, ''ekey''                 *)

new_theory ''pem_clear'';

load_library{lib = hol88_lib, theory = ''-''};
open Psyntax Compat;

new_parent ''pem_syntax'';
new_parent ''pem_definitions'';

add_theory_to_sml ''pem_syntax'';
add_theory_to_sml ''pem_definitions'';

(*=     =       =       =       =       =       =       =       =*)
(* this section is what needed to be redone for a different      *)
(* message structure of MIC_CLEAR                                *)

val micclearmsg = ty_antiq
        (==`:(preeb # proctype # contentdomain # id_asymmetric #
        (certificate list) # MIC_info # string # posteb)`==);

val MIC_CLEAR_example = new_definition
        (''MIC_CLEAR_example'', --`MIC_CLEAR_example s1 s2 s3 s4 =
        (BEGIN ''PRIVACY_ENHANCED MAIL'',
        Proc_Type (4,MIC_CLEAR),
        Content_Domain RFC822,
        ID_Asymmetric (s1:string),
        [Certificate (s2:string)],
        MIC_Info (RSA_MD5,RSA,(s3: string)),
                                (* asymsignmic *)
        (s4: string),
        (* pemtext *)
        END ''PRIVACY_ENHANCED MAIL'')`--);

val get_MIC_CLEAR_MIC_Info = new_definition
        (''get_MIC_CLEAR_MIC_Info'',
        (--`get_MIC_CLEAR_MIC_Info (x: ^micclearmsg) =
          FST(SND(SND(SND(SND x)))))`--));

(* sender ID, this field can replace the sender's certificate   *)
val get_OriginatorAsymID_info = new_definition
        (''get_OriginatorAsymID_info'',
        --`get_OriginatorAsymID_info (x:^micclearmsg)
        = FST(SND(SND(SND x)))`--);


val get_MIC_CLEAR_Proc_Type = new_definition
        (`get_MIC_CLEAR_Proc_Type`,
        (--`get_MIC_CLEAR_Proc_Type (x: ^micclearmsg) = FST(SND x)`--));

val get_MIC_CLEAR_text = new_definition
        (`get_MIC_CLEAR_text`,
        (--`get_MIC_CLEAR_text (x: ^micclearmsg) =
          FST(SND(SND(SND(SND(SND(SND x))))))`--));

(*=      =       =       =       =       =       =       =       ==*)
(* retrieve each sub-field from raw message field               *)
(* these are not used in the following proof, other functions   *)
(* are used instead                                             *)

(* Hash Algorithm                                               *)
val get_msg_HashID = new_definition
        (`get_msg_HashID`,
        (--`get_msg_HashID (x:^micclearmsg) =
        get_MIC_algid (get_MIC_CLEAR_MIC_Info x)`--));

(* Sign Algorithm for message digest                            *)
val get_msg_SignID = new_definition (`get_msg_SignID`,
        --`get_msg_SignID (x:^micclearmsg)
                = get_MIC_sigalgid (get_MIC_CLEAR_MIC_Info x)`--);

(* Encrypted MIC                                                *)
val get_msg_MIC = new_definition
        (`get_msg_MIC`,
        --`get_msg_MIC (x:^micclearmsg)
        = get_MIC_mic (get_MIC_CLEAR_MIC_Info x)`--);


val MIC_CLEAR_is_Intact = new_definition
        (`MIC_CLEAR_is_Intact`,
        (--`MIC_CLEAR_is_Intact (mic_clear_msg:^micclearmsg) =
        (let micInfo = (get_MIC_CLEAR_MIC_Info mic_clear_msg) in
        (let ekey = get_Key_from_ID
            (get_OriginatorAsymID_info mic_clear_msg) in
        (is_Intact
        (MIC_sign_select micInfo)
        (MIC_hash_select micInfo)
        (get_MIC_CLEAR_text mic_clear_msg)
        (get_MIC_mic micInfo) ekey)))`--));

close_theory();
export_theory();

MIC_CLEAR_example;

val integrity_lemma1 = prove_thm
        (`integrity_lemma1`,
        (--`!(verify: string -> string -> string -> bool)
        (sign: string -> string -> string)
        (hash: string -> string)
        (txmessage: string) (rxmessage: string)
        (dkey: string) (ekey: string).
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2. verify m1 (sign m2 dkey) ekey = (m1 = m2)) ==>
        (is_Intact verify hash rxmessage
          (sign (hash txmessage) dkey) ekey)
```

```
            ==> (txmessage = rxmessage)`--),
        REPEAT GEN_TAC THEN
        REWRITE_TAC [is_Intact] THEN
        DISCH_THEN (fn th => ASSUME_TAC (SPECL
          [(--`rxmessage:string`--), (--`txmessage:string`--)] th)) THEN
        DISCH_THEN (fn th => REWRITE_TAC [SPECL
          [(--`(hash:string->string) (rxmessage:string)`--),
           (--`(hash:string->string) (txmessage:string)`--)] th]) THEN
        DISCH_THEN (fn th => ASSUME_TAC th THEN RES_TAC THEN
        ASM_REWRITE_TAC []));

val integrity_lemma2 = prove_thm
        ("integrity_lemma2",
        (--`!(verify: string -> string -> string -> bool)
        (sign: string -> string -> string)
        (hash: string -> string)
        (txmessage: string) (rxmessage: string)
        (dkey: string) (ekey: string).
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2. verify m1 (sign m2 dkey) ekey = (m1 = m2)) ==>
        ((txmessage = rxmessage) ==>
          (is_Intact verify hash rxmessage
            (sign (hash txmessage) dkey) ekey))`--),
        REPEAT GEN_TAC
        THEN REWRITE_TAC [is_Intact]
        THEN DISCH_THEN (fn th1 =>
          (DISCH_THEN (fn th2 => REWRITE_TAC
            [th1,(SPEC (--`(hash:string -> string) txmessage`--)th2)])))
        THEN DISCH_THEN (fn th => REWRITE_TAC [th]));

val integrity_lemma3 = prove_thm
        ("integrity_lemma3",
        (--`!(verify: string -> string -> string -> bool)
        (sign: string -> string -> string)
        (hash: string -> string)
        (txmessage: string) (rxmessage: string)
        (dkey: string) (ekey: string).
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2. verify m1 (sign m2 dkey) ekey = (m1 = m2)) ==>
        ((txmessage = rxmessage) =
          (is_Intact verify hash rxmessage
            (sign (hash txmessage) dkey) ekey))`--),
        REPEAT GEN_TAC
        THEN DISCH_THEN (fn th1 => (DISCH_THEN (fn th2 => EQ_TAC
        THEN MP_TAC th2 THEN MP_TAC th1)))
        THEN REWRITE_TAC [integrity_lemma1,integrity_lemma2]);

 export_theory ();


 (*=     =       =       =       =       =       =       =       =*)
 val Intact = prove_thm ("Intact",
        (--`!(verify: string -> string -> string -> bool)
        (sign: string -> string -> string)
        (hash: string -> string)
        (txmessage: string) (rxmessage: string)
        (dkey: string) (ekey: string)
        (smd:string).
        (smd = (sign (hash txmessage) dkey)) ==>
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2. (verify m1 (sign m2 dkey) ekey) = (m1 = m2)) ==>
              ((txmessage = rxmessage) =
                (is_Intact verify hash rxmessage smd ekey))`--),
        REPEAT GEN_TAC
        THEN DISCH_THEN (fn th => REWRITE_TAC [th])
        THEN REWRITE_TAC [integrity_lemma3]);

fun let_ELIM_CONV t =
        TRY_CONV (let_CONV THENC let_ELIM_CONV) t;

val th1 = let_ELIM_CONV
        (--`let micInfo = get_MIC_CLEAR_MIC_Info mic_clear_msg in
        (let ekey = get_Key_from_ID (get_OriginatorAsymID_info
                mic_clear_msg) in
        (let hash = MIC_hash_select micInfo and
          verify = MIC_sign_select micInfo and
          rxmessage = get_MIC_CLEAR_text mic_clear_msg in
        ((get_MIC_mic micInfo = sign (hash txmessage) dkey) ==>
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2.verify m1
          ((sign:string->string->string) m2 dkey)ekey = (m1 = m2)) ==>
        ((txmessage = rxmessage) =
        MIC_CLEAR_is_Intact mic_clear_msg))))`--);

val th2 = let_ELIM_CONV
        (--`let micInfo = get_MIC_CLEAR_MIC_Info mic_clear_msg
        in
        let ekey = get_Key_from_ID (get_OriginatorAsymID_info mic_clear_msg)
        in
        is_Intact (MIC_sign_select micInfo) (MIC_hash_select micInfo)
          (get_MIC_CLEAR_text mic_clear_msg)
          (get_MIC_mic micInfo)
          ekey`--);

val MIC_CLEAR_is_Intact_Correct = prove_thm
        ("MIC_CLEAR_is_Intact_Correct",
        (--`(!(mic_clear_msg: ^micclearmsg)
        (sign: string -> string-> string)
        (txmessage: string)
        (dkey: string).
        (let micInfo = get_MIC_CLEAR_MIC_Info mic_clear_msg in
        (let ekey = get_Key_from_ID (get_OriginatorAsymID_info
                mic_clear_msg) in
        (let hash = MIC_hash_select micInfo and
          verify = MIC_sign_select micInfo and
          rxmessage = get_MIC_CLEAR_text mic_clear_msg in
        ((get_MIC_mic micInfo = sign (hash txmessage) dkey) ==>
        (!m1 m2.(hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!m1 m2.verify m1 (sign m2 dkey) ekey = (m1 = m2)) ==>
        ((txmessage = rxmessage) =
        MIC_CLEAR_is_Intact mic_clear_msg))))))`--),
  REPEAT GEN_TAC
  THEN REWRITE_TAC [th1]
  THEN REWRITE_TAC [MIC_CLEAR_is_Intact,th2]
  THEN REWRITE_TAC [Intact]);

export_theory();
```

Appendix E

PEM_ENCRYPTED

E.1 pem_encrypted.theory

```
Theory: pem_encrypted

Parents:
    pem_definitions

Type constants:


Term constants:
    ENCRYPTED_example (Prefix)
    :string -> string -> string -> string -> string -> string ->
     preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb
    getEN_DEK_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> dekinfo
    getEN_OriginatorAsymID_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> id_asymmetric
    getEN_IssuerCert_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> certificate list
    getEN_MIC_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> MIC_info
    getEN_KEY_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> Key_info
    getEN_Message_info (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> string
    getEN_msg_MsgEncryptID (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> algid
    getEN_msg_MsgEncryptIV (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
     certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
     posteb -> IV
    getEN_msg_HashID (Prefix)
    :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
```

```
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> algid
     getEN_msg_SignID (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> algid
     getEN_msg_EncryptedMIC (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string
     getEN_msg_KeyEncryptID (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> algid
     getEN_msg_EncryptedKey (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string
     getEN_msg_DEK (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string
     getEN_msg_message (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string
     getEN_msg_MIC (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string
     ENCRYPTED_is_PrivateP (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string -> bool
     ENCRYPTED_is_PrivateS (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> string -> bool
     ENCRYPTED_is_Authentic2 (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> bool
     ENCRYPTED_is_Intact (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> bool
     ENCRYPTED_is_non_deniable (Prefix)
     :preeb # proctype # contentdomain # dekinfo # id_asymmetric #
      certificate list # MIC_info # (id_asymmetric # Key_info) list # string #
      posteb -> bool

Axioms:


Definitions:
    ENCRYPTED_example
    |- !s1 s2 s3 s4 s5 s6.
         ENCRYPTED_example s1 s2 s3 s4 s5 s6 =
         (BEGIN 'PRIVACY_ENHANCED MAIL'',
          Proc_Type (4,ENCRYPTED),
          Content_Domain RFC822,
          DEK_Info (DES_CBC,IV),
          ID_Asymmetric s1,
          [Certificate s2],
        MIC_Info (RSA_MD5,RSA,s3),
        [ID_Asymmetric s4,Key_Info (RSA,s5)],
        s6,
        END 'PRIVACY_ENHANCED MAIL'')
getEN_DEK_info |- !x. getEN_DEK_info x = FST (SND (SND (SND x)))
getEN_OriginatorAsymID_info
|- !x. getEN_OriginatorAsymID_info x = FST (SND (SND (SND (SND x))))
getEN_IssuerCert_info
|- !x. getEN_IssuerCert_info x = FST (SND (SND (SND (SND (SND x)))))
getEN_MIC_info
|- !x. getEN_MIC_info x = FST (SND (SND (SND (SND (SND (SND x))))))
getEN_KEY_info
|- !x.
     getEN_KEY_info x =
     SND (HD (FST (SND (SND (SND (SND (SND (SND (SND x)))))))))
getEN_Message_info
|- !x.
     getEN_Message_info x =
     FST (SND (SND (SND (SND (SND (SND (SND (SND x))))))))
getEN_msg_MsgEncryptID
|- !x. getEN_msg_MsgEncryptID x = get_DEK_algid (getEN_DEK_info x)
getEN_msg_MsgEncryptIV
|- !x. getEN_msg_MsgEncryptIV x = get_DEK_IV (getEN_DEK_info x)
getEN_msg_HashID
|- !x. getEN_msg_HashID x = get_MIC_algid (getEN_MIC_info x)
getEN_msg_SignID
|- !x. getEN_msg_SignID x = get_MIC_sigalgid (getEN_MIC_info x)
getEN_msg_EncryptedMIC
|- !x. getEN_msg_EncryptedMIC x = get_MIC_mic (getEN_MIC_info x)
getEN_msg_KeyEncryptID
|- !x. getEN_msg_KeyEncryptID x = get_KEY_algid (getEN_KEY_info x)
getEN_msg_EncryptedKey
|- !x. getEN_msg_EncryptedKey x = get_KEY_asymsgKey (getEN_KEY_info x)
getEN_msg_DEK
|- !x.
     getEN_msg_DEK x =
     DEK_encrypt_select (getEN_KEY_info x) (getEN_msg_EncryptedKey x)
       recipientkey
getEN_msg_message
|- !x.
     getEN_msg_message x =
     msg_Encrypt_select (getEN_DEK_info x) (getEN_Message_info x)
       (getEN_msg_DEK x)
       (getEN_msg_MsgEncryptIV x)
getEN_msg_MIC
|- !x.
     getEN_msg_MIC x =
     msg_Encrypt_select (getEN_DEK_info x) (getEN_msg_EncryptedMIC x)
       (getEN_msg_DEK x)
       (getEN_msg_MsgEncryptIV x)
ENCRYPTED_is_PrivateP
|- !msg txDEK.
     ENCRYPTED_is_PrivateP msg txDEK =
     is_PrivateP (DEK_encrypt_select (getEN_KEY_info msg)) txDEK
       (getEN_msg_EncryptedKey msg)
       recipientkey
ENCRYPTED_is_PrivateS
|- !msg message.
     ENCRYPTED_is_PrivateS msg message =
     (let rxDEK = getEN_msg_DEK msg
      and
      decryptIV = getEN_msg_MsgEncryptIV msg
      in
      is_PrivateS (msg_Encrypt_select (getEN_DEK_info msg)) message
```

```
         (getEN_Message_info msg)
         decryptIV
         rxDEK)
  ENCRYPTED_is_Authentic2
  |- !msg.
       ENCRYPTED_is_Authentic2 msg =
       (let micInfo = getEN_MIC_info msg
        in
        let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        is_Authentic2 (MIC_sign_select micInfo) (MIC_hash_select micInfo)
          (getEN_msg_message msg)
          (getEN_msg_MIC msg)
          ekey)
  ENCRYPTED_is_Intact
  |- !msg.
       ENCRYPTED_is_Intact msg =
       (let micInfo = getEN_MIC_info msg
        in
        let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        is_Intact (MIC_sign_select micInfo) (MIC_hash_select micInfo)
          (getEN_msg_message msg)
          (getEN_msg_MIC msg)
          ekey)
  ENCRYPTED_is_non_deniable
  |- !msg.
       ENCRYPTED_is_non_deniable msg =
       (let micInfo = getEN_MIC_info msg
        in
        let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        and
        hash = MIC_hash_select micInfo
        in
        is_non_deniable (MIC_sign_select micInfo)
          (hash (getEN_msg_message msg))
          (getEN_msg_MIC msg)
          ekey)

Theorems:
  ENCRYPTED_is_Private_DEK
  |- !Encrypted_msg encryptP DEK dKEY0 dkey.
       let Key_info = getEN_KEY_info Encrypted_msg
       in
       let decryptP = DEK_encrypt_select Key_info
       and
       rxmsg = getEN_msg_EncryptedKey Encrypted_msg
       and
       dkey = recipientkey
       in
       (rxmsg = txmsg) ==>
       (txmsg = encryptP DEK ekey) ==>
       (!msg. decryptP (encryptP msg ekey) dKEY0 = msg) ==>
       (!msg d2.
         (decryptP (encryptP msg ekey) d2 = msg) ==> (d2 = dKEY0)) ==>
       ((dkey = dKEY0) = ENCRYPTED_is_PrivateP Encrypted_msg DEK)
  ENCRYPTED_is_Private_msg
  |- !Encrypted_msg encryptS message DEK.
       let DEK_info = getEN_DEK_info Encrypted_msg
       in
       let decryptS = msg_Encrypt_select DEK_info
       and
       rxmsg = getEN_Message_info Encrypted_msg
       and
```

```
      decryptIV = getEN_msg_MsgEncryptIV Encrypted_msg
      and
      KEY0 = DEK
      and
      key = getEN_msg_DEK Encrypted_msg
      in
      (rxmsg = txmsg) ==>
      (txmsg = encryptS message KEY0 decryptIV) ==>
      (!msg key.
        (decryptS (encryptS msg key decryptIV) key decryptIV = msg) /\
        (!msg key1.
          (decryptS msg key1 decryptIV = decryptS msg key decryptIV) =
          key =
          key1)) ==>
      ((key = KEY0) = ENCRYPTED_is_PrivateS Encrypted_msg message)
ENCRYPTED_is_Authentic_msg
|- !Encrypted_msg sign txmic dKEY0 dkey.
      let micInfo = getEN_MIC_info Encrypted_msg
      in
      let verify = MIC_sign_select micInfo
      and
      hash = MIC_hash_select micInfo
      and
      message = getEN_msg_message Encrypted_msg
      and
      rxmic = getEN_msg_MIC Encrypted_msg
      and
      ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
      in
      (rxmic = txmic) ==>
      (txmic = sign (hash message) dkey) ==>
      (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey = dkey2 = dKEY0) ==>
      ((dkey = dKEY0) = ENCRYPTED_is_Authentic2 Encrypted_msg)
ENCRYPTED_is_Intact_msg
|- !Encrypted_msg sign txmessage txmic dkey.
      let micInfo = getEN_MIC_info Encrypted_msg
      in
      let verify = MIC_sign_select micInfo
      and
      hash = MIC_hash_select micInfo
      and
      rxmessage = getEN_msg_message Encrypted_msg
      and
      rxmic = getEN_msg_MIC Encrypted_msg
      and
      ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
      in
      (txmic = sign (hash txmessage) dkey) ==>
      (rxmic = txmic) ==>
      (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
      (!s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ==>
      ((rxmessage = txmessage) = ENCRYPTED_is_Intact Encrypted_msg)
ENCRYPTED_is_non_deniable_msg
|- !Encrypted_msg sign MESSAGE0 txmic dKEY0 dkey.
      let micInfo = getEN_MIC_info Encrypted_msg
      in
      let verify = MIC_sign_select micInfo
      and
      hash = MIC_hash_select micInfo
      and
      message = getEN_msg_message Encrypted_msg
      and
      rxmic = getEN_msg_MIC Encrypted_msg
      and
```

```
        ekey = get_Key_from_ID (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ==>
        (txmic = sign (hash MESSAGE0) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
        (!m1 m2 dkey2.
          verify m1 (sign m2 dkey2) ekey = (m1 = m2) /\ (dkey2 = dKEY0)) ==>
        ((dkey = dKEY0) /\ (message = MESSAGE0) =
         ENCRYPTED_is_non_deniable Encrypted_msg)
```

E.2 pem_encrypted.sml

```
(*==================================================*)
(* File:        pem_encrypted.sml                   *)
(* Description: selector and security function for  *)
(*              ENCRYPTED message                   *)
(* Date:        Aug. 20, 1996                       *)
(* Author:      Dan Zhou                            *)
(*==================================================*)


new_theory ''pem_encrypted'';

load_library{lib = hol88_lib, theory = ''-''};
open Psyntax Compat;

new_parent ''pem_syntax'';
new_parent ''pem_definitions'';

add_theory_to_sml ''pem_syntax'';
add_theory_to_sml ''pem_definitions'';

(*=     =       =       =       =       =       =       =       =*)
(* abbreviated PEM Message type *)
val encryptedmsg = ty_antiq
        (==`:(preeb#proctype#contentdomain#dekinfo#id_asymmetric
        #(certificate list)#MIC_info#(id_asymmetric#Key_info)list
        #string#posteb)`==);
        (* pemtext *)


(*=     =       =       =       =       =       =       =       =*)
val ENCRYPTED_example = new_definition
        (''ENCRYPTED_example'', --`ENCRYPTED_example s1 s2 s3 s4 s5 s6=
        (BEGIN ''PRIVACY_ENHANCED MAIL'',
        Proc_Type (4,ENCRYPTED),
        Content_Domain RFC822,
        DEK_Info (DES_CBC,(IV:IV)),
        ID_Asymmetric (s1:string),
        [Certificate (s2:string)],
        MIC_Info (RSA_MD5,RSA,(s3:string)),
                                (* asymsignmic *)
        [(ID_Asymmetric (s4:string), Key_Info (RSA, (s5:string)))],
                                                (* asymsgKey *)
        (s6:string),
        (* pemtext *)
        END ''PRIVACY_ENHANCED MAIL'')`--);


(*=     =       =       =       =       ==      =       =       =*)
(* retrieve raw fields from received PEM-Encrypted-Message      *)
(* without any operation                                          *)

(* Mesasge Encryption Alrogithm, and IV                           *)
val getEN_DEK_info : new_definition ("getEN_DEK_info",
        (--`getEN_DEK_info (x:^encryptedmsg)
        : FST(SND(SND(SND x)))`--));

(* sender ID, this field can replace the sender's certificate    *)
val getEN_OriginatorAsymID_info : new_definition
        ("getEN_OriginatorAsymID_info",
        --`getEN_OriginatorAsymID_info (x:^encryptedmsg)
        : FST(SND(SND(SND(SND x))))`--);

(* CA certificate                                                 *)
val getEN_IssuerCert_info : new_definition
        ("getEN_IssuerCert_info",
        --`getEN_IssuerCert_info (x:^encryptedmsg)
        :  FST(SND(SND(SND(SND(SND x)))))`--);

(* Message Digest Algorithm, Message Digest Sign Algorithm,       *)
(* *encrypted* MIC                                                *)
val getEN_MIC_info : new_definition ("getEN_MIC_info",
        (--`getEN_MIC_info (x:^encryptedmsg)
        : FST(SND(SND(SND(SND(SND(SND x))))))`--));

(* recipient ID.  For recipient's certificate                     *)
(* this will not be used until later
val getEN_Recipients_info :  new_definition ("getEN_Recipients_info",
        --`getEN_Recipients_info  (x:^encryptedmsg)
        : FST(SND(SND(SND(SND(SND(SND(SND x)))))))`--);
*)

(* Recipient ID: this is used to get the public/private key      *)
(* of recipient                                                   *)
(* ------------ this is not used right now ----------------      *)
(* we just assume recipient publickey and private key is         *)
(* available                                                      *)


(* Recipient Key-info: per-message key encryption Algorithm      *)
(* and Encrypted per-message key                                  *)
(* this will be used temporarily *)
val getEN_KEY_info : new_definition ("getEN_KEY_info",
        (--`getEN_KEY_info (x:^encryptedmsg)
        : SND(HD (FST(SND(SND(SND(SND(SND(SND(SND x)))))))))`--));

(* the encrypted message                                          *)
val getEN_Message_info : new_definition ("getEN_Message_info",
        --`getEN_Message_info (x:^encryptedmsg)
        : FST(SND(SND(SND(SND(SND(SND(SND x))))))))`--);


(*:      :       :       :       :       :       :       :       :*)
(* retrieve each individual sub-field from raw message field     *)

(* Message encryption Algorithm                                   *)
val getEN_msg_MsgEncryptID : new_definition
        ("getEN_msg_MsgEncryptID",
                --`getEN_msg_MsgEncryptID (x:^encryptedmsg)
                : get_DEK_algid (getEN_DEK_info x)`--);

(* Message encryption IV                                          *)
val getEN_msg_MsgEncryptIV : new_definition
        ("getEN_msg_MsgEncryptIV",
                --`getEN_msg_MsgEncryptIV (x:^encryptedmsg)
                = get_DEK_IV (getEN_DEK_info x)`--);


(*=     =       =       =       =       =       =       =       =*)
(* Hash Algorithm                                                *)
val getEN_msg_HashID = new_definition (``getEN_msg_HashID'',
        --`getEN_msg_HashID (x:^encryptedmsg)
                = get_MIC_algid (getEN_MIC_info x)`--);

(* Sign Algorithm for message digest                             *)
val getEN_msg_SignID = new_definition (``getEN_msg_SignID'',
        --`getEN_msg_SignID (x:^encryptedmsg)
                = get_MIC_sigalgid (getEN_MIC_info x)`--);

(* Encrypted MIC                                                 *)
val getEN_msg_EncryptedMIC = new_definition
        (``getEN_msg_EncryptedMIC'',
        --`getEN_msg_EncryptedMIC (x:^encryptedmsg)
        = get_MIC_mic (getEN_MIC_info x)`--);


(*=     =       =       =       =       =       =       =       =*)
(* message key encryption Algorithm                              *)
val getEN_msg_KeyEncryptID = new_definition (``getEN_msg_KeyEncryptID'',
        --`getEN_msg_KeyEncryptID (x:^encryptedmsg)
        = get_KEY_algid (getEN_KEY_info x)`--);

(* Encrypted Message Key                                         *)
val getEN_msg_EncryptedKey = new_definition (``getEN_msg_EncryptedKey'',
        --`getEN_msg_EncryptedKey (x:^encryptedmsg)
        = get_KEY_asymsgKey (getEN_KEY_info x)`--);


(*=     =       =       =       =       =       =       =       =*)
(* extract DEK/original message/MIC from the received message   *)
val getEN_msg_DEK = new_definition (``getEN_msg_DEK'',
        --`getEN_msg_DEK (x:^encryptedmsg)
        = (DEK_encrypt_select (getEN_KEY_info x))
        (getEN_msg_EncryptedKey x) recipientkey`--);

val getEN_msg_message = new_definition (``getEN_msg_message'',
        --`getEN_msg_message (x:^encryptedmsg)
        = (msg_Encrypt_select (getEN_DEK_info x))
        (getEN_Message_info x) (getEN_msg_DEK x)
        (getEN_msg_MsgEncryptIV x)`--);

(* notice here the IV is the same as message encryptiong IV    *)
(* this is my assumption                                        *)
val getEN_msg_MIC = new_definition (``getEN_msg_MIC'',
        --`getEN_msg_MIC (x:^encryptedmsg)
        = (msg_Encrypt_select (getEN_DEK_info x))
        (getEN_msg_EncryptedMIC x)
        (getEN_msg_DEK x) (getEN_msg_MsgEncryptIV x)`--);


(*=     =       =       =       =       =       =       =       =*)
(* Define security functions for PEM-Encrypted-Message         *)

(* by this, we test the DEK is private.                         *)
val ENCRYPTED_is_PrivateP = new_definition (``ENCRYPTED_is_PrivateP'',
        --`ENCRYPTED_is_PrivateP (msg: ^encryptedmsg) (txDEK:string)
        = is_PrivateP (DEK_encrypt_select (getEN_KEY_info msg))
        txDEK (getEN_msg_EncryptedKey msg) recipientkey`--);
```

```
(* by this, we test the msg is private.                              *)
val ENCRYPTED_is_PrivateS : new_definition ('ENCRYPTED_is_PrivateS'',
        --'ENCRYPTED_is_PrivateS (msg: ^encryptedmsg) (message:string)
        : let rxDEK : getEN_msg_DEK msg
        and
        decryptIV : getEN_msg_MsgEncryptIV msg
        in
        (is_PrivateS (msg_Encrypt_select (getEN_DEK_info msg))
        message (getEN_Message_info msg) decryptIV rxDEK)'--);


(* test for message authentication, no need to use the other         *)
(* form                                                              *)
val ENCRYPTED_is_Authentic2 : new_definition (
        'ENCRYPTED_is_Authentic2'',
        --'ENCRYPTED_is_Authentic2 (msg: ^encryptedmsg)
        :(let micInfo : getEN_MIC_info msg in
        (let ekey : get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        (is_Authentic2 (MIC_sign_select micInfo)
        (MIC_hash_select micInfo) (getEN_msg_message msg)
        (getEN_msg_MIC msg) ekey)))'--);


(* by this, we test the message that Originator sent is intact  *)
val ENCRYPTED_is_Intact : new_definition ('ENCRYPTED_is_Intact'',
        (--'ENCRYPTED_is_Intact (msg: ^encryptedmsg)
        : (let micInfo : getEN_MIC_info msg in
        (let ekey : get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        (is_Intact (MIC_sign_select micInfo)
        (MIC_hash_select micInfo) (getEN_msg_message msg)
        (getEN_msg_MIC msg) ekey)))'--));



(* test for message non-deniability                                  *)
val ENCRYPTED_is_non_deniable : new_definition (
        'ENCRYPTED_is_non_deniable'',
        --'ENCRYPTED_is_non_deniable (msg: ^encryptedmsg)
        : (let micInfo : getEN_MIC_info msg in
        (let ekey : get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        and
          hash : MIC_hash_select micInfo
        in
        (is_non_deniable (MIC_sign_select micInfo)
        (hash (getEN_msg_message msg))
        (getEN_msg_MIC msg) ekey)))'--);

close_theory();
export_theory();


(*:       :         :         :         :         :         :         :        :*)
(* prove properties of Encrypted PEM message                          *)

fun let_ELIM_CONV t :
        TRY_CONV (let_CONV THENC let_ELIM_CONV) t;


(* 1. ENCRYPTED_is_Private_DEK                                        *)
(* recipientkey: the private key of recipient                         *)
(* ekey:          public key of the intended recipient                *)
(* dKEY0:        private key of the intended recipient         *)


val th1 = let_ELIM_CONV
        (--`let Key_info = getEN_KEY_info Encrypted_msg in
        let decryptP = DEK_encrypt_select Key_info
        and
          rxmsg = getEN_msg_EncryptedKey Encrypted_msg
        and
          dkey = recipientkey
        in
        (rxmsg = txmsg) ==>
        (txmsg = encryptP DEK (ekey:string)) ==>
        (!msg. decryptP (encryptP msg ekey) dKEY0 = msg) ==>
        (!msg d2. (decryptP (encryptP msg ekey) d2 = msg)
          ==> (d2 = dKEY0)) ==>
        ((dkey = dKEY0) = ENCRYPTED_is_Private
                Encrypted_msg (DEK:string))`--);


(* --- is there a need to have dkey=recipient ---              *)

val ENCRYPTED_is_Private_DEK = prove_thm
        ("ENCRYPTED_is_Private_DEK",
        --`!(Encrypted_msg: ^encryptedmsg)
        (encryptP: string->string->string)
        (DEK: string) (dKEY0: string) (dkey: string).
        let Key_info = getEN_KEY_info Encrypted_msg in
        let decryptP = DEK_encrypt_select Key_info
        and
          rxmsg = getEN_msg_EncryptedKey Encrypted_msg
        and
          dkey = recipientkey
        in
        (rxmsg = txmsg) ==>
        (txmsg = encryptP DEK ekey) ==>
        (!msg. decryptP (encryptP msg ekey) dKEY0 = msg) ==>
        (!msg d2. (decryptP (encryptP msg ekey) d2 = msg)
          ==> (d2 = dKEY0)) ==>
        ((dkey = dKEY0)
                = ENCRYPTED_is_PrivateP Encrypted_msg DEK)`--,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [th1] THEN
        REWRITE_TAC [ENCRYPTED_is_PrivateP] THEN
        ACCEPT_TAC (SPECL
          [--`DEK_encrypt_select (getEN_KEY_info Encrypted_msg)`--,
          --`encryptP: string -> string -> string`--,
          --`DEK: string`--,
          --`txmsg: string`--,
          --`getEN_msg_EncryptedKey Encrypted_msg`--,
          --`ekey: string`--,
          --`dKEY0: string`--,
          --`recipientkey: string`--] is_Private_DEK));


(*=     =       =       =       =       =       =       =       =*)
(* 2. ENCRYPTED_is_Private_msg                                  *)
val th1 = let_ELIM_CONV (
        --`let DEK_info = getEN_DEK_info Encrypted_msg in
        let decryptS = msg_Encrypt_select DEK_info
        and
          rxmsg = getEN_Message_info Encrypted_msg
        and
          decryptIV = getEN_msg_MsgEncryptIV Encrypted_msg
        and
          KEY0 = DEK
        and
          key = getEN_msg_DEK Encrypted_msg
        in
        (rxmsg = txmsg) ==>
        (txmsg = encryptS message KEY0 decryptIV) ==>
        (!msg key. (decryptS (encryptS msg key decryptIV)
                key decryptIV = msg) /\
        !msg key1. ((decryptS msg key1 decryptIV =
                decryptS msg key decryptIV) =  key = key1)) ==>
        ((key = KEY0)
                = ENCRYPTED_is_PrivateS Encrypted_msg message)`--);

val th2 = let_ELIM_CONV (
        --`let rxDEK = getEN_msg_DEK msg
        and
        decryptIV = getEN_msg_MsgEncryptIV msg
        in
        is_PrivateS (msg_Encrypt_select (getEN_DEK_info msg)) message
          (getEN_Message_info msg)
          decryptIV
          rxDEK`--);
val th3 = REWRITE_RULE [th2] ENCRYPTED_is_PrivateS;

val ENCRYPTED_is_Private_msg = prove_thm
        ("ENCRYPTED_is_Private_msg",
        --`!(Encrypted_msg: ^encryptedmsg)
        (encryptS: string->string->IV->string)
        (message: string) (DEK: string).
        let DEK_info = getEN_DEK_info Encrypted_msg in
        let decryptS = msg_Encrypt_select DEK_info
        and
          rxmsg = getEN_Message_info Encrypted_msg
        and
          decryptIV = getEN_msg_MsgEncryptIV Encrypted_msg
        and
          KEY0 = DEK
        and
          key = getEN_msg_DEK Encrypted_msg
        in
        (rxmsg = txmsg) ==>
        (txmsg = encryptS message KEY0 decryptIV) ==>
        (!msg key. (decryptS (encryptS msg key decryptIV)
                key decryptIV = msg) /\
        !msg key1. ((decryptS msg key1 decryptIV =
                decryptS msg key decryptIV) =  key = key1)) ==>
        ((key = KEY0)
                = ENCRYPTED_is_PrivateS Encrypted_msg message)`--,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [th1] THEN
        REWRITE_TAC [th3] THEN
        ACCEPT_TAC (SPECL
         [--`msg_Encrypt_select (getEN_DEK_info Encrypted_msg)`--,
          --`encryptS: string -> string -> IV -> string`--,
          --`message: string`--,
          --`txmsg: string`--,
          --`getEN_Message_info Encrypted_msg`--,
          --`getEN_msg_MsgEncryptIV Encrypted_msg`--,
          --`DEK:string`--,
          --`getEN_msg_DEK Encrypted_msg`--] is_Private_msg));


(*=     =       =       =       =       =       =       =       =*)
```

```
(* 3. ENCRYPTED_is_Authentic_msg                            *)
(* dkey: originator's private key                           *)
(* dKEY0: the key of the one we think who sent the mail     *)
(* ekey:  the public key of the one who we think sent the mail  *)
(* since ekey is publicly known.  --- May need more work ---    *)

val th1 = let_ELIM_CONV
        (--`let micInfo = getEN_MIC_info Encrypted_msg in
        let verify = MIC_sign_select micInfo
        and
          hash = MIC_hash_select micInfo
        and
          message = getEN_msg_message Encrypted_msg
        and
          rxmic = getEN_msg_MIC Encrypted_msg
        and
          ekey = get_Key_from_ID
            (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ==>
        (txmic = (sign:string->string->string) (hash message) dkey) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = dkey2 = dKEY0) ==>
        ((dkey = dKEY0) =
                ENCRYPTED_is_Authentic2 Encrypted_msg)`--);

val th2 = let_ELIM_CONV
        (--`let micInfo = getEN_MIC_info msg in
        (let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        (is_Authentic2 (MIC_sign_select micInfo)
        (MIC_hash_select micInfo) (getEN_msg_message msg)
        (getEN_msg_MIC msg) ekey))`--);

val th3 = REWRITE_RULE [th2] ENCRYPTED_is_Authentic2;


val ENCRYPTED_is_Authentic_msg = prove_thm
        ("ENCRYPTED_is_Authentic_msg",
        --`!(Encrypted_msg:^encryptedmsg)
        (sign: string -> string -> string)  (txmic:string)
        (dKEY0:string) (dkey:string).
        let micInfo = getEN_MIC_info Encrypted_msg in
        let verify = MIC_sign_select micInfo
        and
          hash = MIC_hash_select micInfo
        and
          message = getEN_msg_message Encrypted_msg
        and
          rxmic = getEN_msg_MIC Encrypted_msg
        and
          ekey = get_Key_from_ID
            (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ==>
        (txmic = (sign:string->string->string) (hash message) dkey) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = dkey2 = dKEY0) ==>
        ((dkey = dKEY0) =
                ENCRYPTED_is_Authentic2 Encrypted_msg)`--,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [th1] THEN
        REWRITE_TAC [th3] THEN
        ACCEPT_TAC (SPECL
        [--`MIC_sign_select (getEN_MIC_info Encrypted_msg)`--,
        --`sign: string -> string -> string`--,
        --`MIC_hash_select (getEN_MIC_info Encrypted_msg)`--,
        --`getEN_msg_message Encrypted_msg`--,
        --`txmic: string`--,
        --`getEN_msg_MIC Encrypted_msg`--,
        --`get_Key_from_ID
              (getEN_OriginatorAsymID_info Encrypted_msg)`--,
        --`dKEY0: string`--,
        --`dkey: string`--] is_Authentic_msg));


(*=     =       =       =       =       =       =       =       =*)
(* 4. ENCRYPTED_is_Intact_msg                                    *)

val th1 = let_ELIM_CONV (--`let micInfo = getEN_MIC_info Encrypted_msg
        in
        let verify = MIC_sign_select micInfo
        and
        hash = MIC_hash_select micInfo
        and
        rxmessage = getEN_msg_message Encrypted_msg
        and
        rxmic = getEN_msg_MIC Encrypted_msg
        and
        ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        (txmic = (sign:string->string->string) (hash txmessage) dkey) ==>
        (rxmic = txmic) ==>
        (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ==>
        ((rxmessage = txmessage) = ENCRYPTED_is_Intact Encrypted_msg)`--);

val th2 = let_ELIM_CONV (--`let micInfo = getEN_MIC_info msg
        in
        let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        in
        is_Intact (MIC_sign_select micInfo) (MIC_hash_select micInfo)
          (getEN_msg_message msg)
          (getEN_msg_MIC msg) ekey`--);
val th3 = REWRITE_RULE [th2] ENCRYPTED_is_Intact;



val ENCRYPTED_is_Intact_msg = prove_thm ("ENCRYPTED_is_Intact_msg",
        --`!(Encrypted_msg: ^encryptedmsg)
        (sign: string->string->string)
        (txmessage: string) (txmic:string) (dkey: string).
        let micInfo = getEN_MIC_info Encrypted_msg in
        let verify = MIC_sign_select micInfo
        and
          hash = MIC_hash_select micInfo
        and
          rxmessage = getEN_msg_message Encrypted_msg
        and
          rxmic = getEN_msg_MIC Encrypted_msg
        and
          ekey = get_Key_from_ID
            (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (txmic = sign (hash txmessage) dkey) ==>
        (rxmic = txmic) ==>
        (!m1 m2. (hash m1 = hash m2) ==> (m1 = m2)) ==>
        (!s1 s2. verify s1 (sign s2 dkey) ekey = s1 = s2) ==>
```

```
       ((rxmessage = txmessage) = ENCRYPTED_is_Intact Encrypted_msg)`--,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [th1] THEN
        REWRITE_TAC [th3] THEN
        ACCEPT_TAC (SPECL
        [--`MIC_sign_select (getEN_MIC_info (Encrypted_msg:^encryptedmsg))`--,
        --`sign:string->string->string`--,
        --`MIC_hash_select (getEN_MIC_info (Encrypted_msg:^encryptedmsg))`--,
        --`txmessage:string`--,
        --`getEN_msg_message (Encrypted_msg:^encryptedmsg)`--,
        --`txmic: string`--,
        --`getEN_msg_MIC (Encrypted_msg:^encryptedmsg)`--,
        --`get_Key_from_ID
          (getEN_OriginatorAsymID_info (Encrypted_msg:^encryptedmsg))`--,
        --`dkey:string`--]
        is_Intact_msg));


(*:      :       :       :       :       :       :       :       :*)
(* 5. ENCRYPTED_is_non_deniable_msg                              *)

val th1 = let_ELIM_CONV (
        --`let micInfo = getEN_MIC_info Encrypted_msg in
        let verify = MIC_sign_select micInfo
        and
          hash = MIC_hash_select micInfo
        and
          message = getEN_msg_message Encrypted_msg
        and
          rxmic = getEN_msg_MIC Encrypted_msg
        and
          ekey = get_Key_from_ID
            (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ==>
        (txmic = (sign: string->string->string)
                (hash MESSAGE0) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = (m1 = m2) /\ (dkey2 = dKEY0)) ==>
        ((dkey = dKEY0) /\ (message = MESSAGE0) =

ENCRYPTED_is_non_deniable Encrypted_msg)`--);

val th2 = let_ELIM_CONV (
        --`let micInfo = getEN_MIC_info msg
        in
        let ekey = get_Key_from_ID (getEN_OriginatorAsymID_info msg)
        and
        hash = MIC_hash_select micInfo
        in
        is_non_deniable (MIC_sign_select micInfo)
          (hash (getEN_msg_message msg)) (getEN_msg_MIC msg) ekey`--);

val th3 = REWRITE_RULE [th2] ENCRYPTED_is_non_deniable;

val ENCRYPTED_is_non_deniable_msg = prove_thm
        ("ENCRYPTED_is_non_deniable_msg",
        --`!(Encrypted_msg: ^encryptedmsg)
        (sign: string -> string ->string) MESSAGE0 txmic dKEY0 dkey.
        let micInfo = getEN_MIC_info Encrypted_msg in
        let verify = MIC_sign_select micInfo
        and
          hash = MIC_hash_select micInfo
        and
          message = getEN_msg_message Encrypted_msg
        and
          rxmic = getEN_msg_MIC Encrypted_msg
        and
          ekey = get_Key_from_ID
            (getEN_OriginatorAsymID_info Encrypted_msg)
        in
        (rxmic = txmic) ==>
        (txmic = sign (hash MESSAGE0) dkey) ==>
        (!m1 m2. (hash m1 = hash m2) = m1 = m2) ==>
        (!m1 m2 dkey2. verify m1 (sign m2 dkey2) ekey
                = (m1 = m2) /\ (dkey2 = dKEY0)) ==>
        ((dkey = dKEY0) /\ (message = MESSAGE0) =
                ENCRYPTED_is_non_deniable Encrypted_msg)`--,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [th1] THEN
        REWRITE_TAC [th3] THEN
        ACCEPT_TAC (SPECL
        [--`MIC_sign_select (getEN_MIC_info Encrypted_msg)`--,
        --`sign: string -> string -> string`--,
        --`MIC_hash_select (getEN_MIC_info Encrypted_msg)`--,
        --`getEN_msg_message Encrypted_msg`--,
        --`MESSAGE0: string`--,
        --`txmic: string`--,
        --`getEN_msg_MIC Encrypted_msg`--,
        --`get_Key_from_ID
                (getEN_OriginatorAsymID_info Encrypted_msg)`--,
        --`dKEY0: string`--,
        --`dkey: string`--] is_non_deniable_msg));

export_theory();
```

Bibliography

[1] D. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. RFC 1423, TIS, February 1993. ftp: ds.internic.net.

[2] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security Private Communication in a Public World.* Prentice Hall, New Jersey, 1995.

[3] David H. Crocker. Standard for the Format of ARPA Internet Text Messages. RFC 822, University of Delaware, August 1982. ftp: ds.internic.net.

[4] John P. Van Tassel D. Randolf Johnson, Fay F. Saydjari. MISSI Security Policy: A Formal Approach. Technical Report R2SPO-TR001-95, INFOSEC Research and Technology Group, National Security Agency, July 1995.

[5] M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis.* Kluwer, 1987.

[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[7] B. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC 1424, RSA Laboratories, February 1993. ftp: ds.internic.net.

[8] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC 1422, BBN, February 1993. ftp: ds.internic.net.

[9] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, DEC, February 1993. ftp: ds.internic.net.

[10] Christopher S. Marron. A PROMELA Model of the MISSI Architecture. Technical Report R2SPO-TR002-95, INFOSEC Research and Technology Group, National Security Agency, 21 September 1995.

[11] T. Melham. Automating Recursive Type Definitions in Higher Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[12] Spyrus, 2814 Junction Ave, Suite 110, San Jose, CA. *FORTEZZA Application Implementors Guide*, 30 January 1995.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61242